Aplicación de la ingeniería software sobre la herramienta MATE *Analyzer*

Memòria del projecte
d'Enginyeria Tècnica en
Informàtica de sistemes
realitzat per
Rodrigo Echeverría Correa
i dirigit per
Anna Sikora

Escola d'Enginyeria
Sabadell, *septiembre* de *2011*

Agradecimientos

Primero de todo, debo dar las gracias a mis dos compañeros, Noel De Martin y Toni Pimenta, sin los cuales este proyecto hubiese sido imposible. Además me gustaría agradecerles no solo este proyecto, si no todos y cada uno en los que hemos colaborado durante la carrera.

También me gustaría agradecer a los directores de este proyecto, Anna Sikora y Eduardo Cesar por acordarse de nosotros y darnos la oportunidad para llevar este proyecto tan interesante. Gracias por depositar vuestra confianza en este equipo.

Además, he de agradecer a Joan Piedrafita, nuestro asesor en este proyecto, su soporte, sus ideas, su contribución y sus sugerencias, sin los cuales este proyecto no podría haberse llevado a cabo.

Por otra parte, me gustaría dar las gracias a mi familia por estar siempre ahí preocupándose y dándome su apoyo en todas las locuras que hago.

Por último, y no menos importantes, debo dar las gracias a todas las personas a las que, en los últimos años, he tenido que decir “Hoy no puedo, tengo cosas que hacer”, lo siento, ya sabéis como soy, quedaré con todos vosotros para celebrar esto en compensación.

FULL DE RESUM
PROJECTE FI DE CARRERA DE L'ESCOLA D'ENGINYERIA

Títol del projecte: Aplicación de la ingeniería software sobre la herramienta MATE: Analyzer	
Autor: Rodrigo Echeverría Correa	**Data:** setembre de 2011
Tutora: Anna Sikora	
Titulació: Enginyeria tècnica en informàtica de sistemes	
Paraules clau (mínim 3) • Català: qualitat de software, MATE, metodologia, entorn de desenvolupament. • Castellà: calidad de software, MATE, metodología, entorno de desarrollo. • Anglès: software quality , MATE, methodology, development environment.	

Resum del projecte (extensió màxima 100 paraules)

- Català:

  MATE (Monitoring, Analysis and Tuning Environment) és un projecte que sorgeix al 2004 com a tesi doctoral de Anna Sikora amb el propòsit de investigar la millora del rendiment d'aplicacions paral·leles a traves de la modificació dinàmica.

  El nostre projecte suposa un pas endavant en termes de qualitat de software i pretén dotar al projecte MATE de una base de desenvolupament sòlida de cara a futures línies de treball. Per això es fa front a la problemàtica des de tres perspectives: la creació d'una metodologia de desenvolupament (i la seva aplicació al projecte existent), la implantació d'un entorn de desenvolupament de suport i el desenvolupament de noves característiques per afavorir la portabilitat i la usabilitat entre altres aspectes.

- Castellà:

  MATE (Monitoring, Analysis and Tuning Environment) es un proyecto que surge en 2004 como tesis doctoral de Anna Sikora con el propósito de investigar la mejora de rendimiento de aplicaciones paralelas a través de la modificación dinámica.

  Nuestro proyecto supone un paso adelante en cuestiones de calidad de software y pretende dotar al proyecto MATE de una base de desarrollo sólida de cara a futuras lineas de trabajo. Para ello se hace frente a la problemática desde tres perspectivas: la creación de una metodología de desarrollo (y su aplicación sobre el proyecto existente), la implantación de un entorno de desarrollo de soporte y el desarrollo de nuevas características para favorecer la portabilidad y la usabilidad, entre otros aspectos.

- Anglès:

  MATE (Monitoring, Analysis and Tuning Environment) is a project that came up in 2004 as the Anna Sikora Ph.D. thesis with the purpose of investigating in the improvement of parallel application's performance through dynamic tuning. Our project means a step forward in software quality concepts and expects to provide the MATE project with a solid development base for the forthcoming lines of work. With this purpose, we face the problem in three ways: the creation of a development methodology (and its implementation in the current project), the deployment of a support development environment and the implementation of new features to improve the portability and the usability among other aspects.

Índice de contenidos

1. Introducción

En este capítulo se exponen los antecedentes al proyecto MATE y el papel que juega este en el marco de la computación de altas prestaciones. Una vez introducido el proyecto MATE, se detalla la contribución del presente proyecto al mismo, sus objetivos y su alcance. Para acabar, se presenta la organización de esta memoria.

1.1. Perspectiva general

Para comenzar, cabe decir que este es un proyecto dedicado a la calidad del software para una aplicación que está enmarcada en la rama de la computación de altas prestaciones. Puesto que en capítulos posteriores se requiere de muchos conceptos de esta rama, el principio de esta introducción se dedica a presentar este marco y el papel que juega MATE en él.

1.1.1. Computación de altas prestaciones

La computación de altas prestaciones (High Performance Computing, HPC, en inglés) es una rama de las ciencias de la computación cuyo objetivo es la creación de sistemas de alto rendimiento capaces de afrontar problemas complejos. Para ello, este campo agrupa diversas tecnologías tanto hardware (electrónica digital, arquitectura de computadores, redes de computadores) como software (algoritmia, software base, sistemas operativos). *[WIKI 11]*

En las últimas décadas esta clase de tecnologías ha cobrado especial relevancia debido a que otros campos de la ciencia como la física, la química o la medicina han incrementado sus necesidades de cálculo y de explotación de información en masa.

Para cubrir estas necesidades de procesamiento se ha optado por aprovechar la capacidad de división de los problemas para computar su solución en sistemas paralelos. Existen básicamente dos fuentes de paralelismo explotables, paralelismo de datos y paralelismo de tareas:

- El paralelismo de datos consiste en dividir grandes estructuras de datos sobre los que se hacen operaciones similares entre las unidades de cálculo.
- El paralelismo de tareas consiste en dividir el programa en tareas independientes para que sean ejecutadas al mismo tiempo en procesadores diferentes.

Respecto a la arquitectura, los sistemas paralelos se caracterizan por poseer varias unidades de cálculo capaces de explotar múltiples instrucciones sobre múltiples flujos de datos (MIMD, Multiple Instruction, Multiple Data). Esta clase de sistemas se pueden clasificar en dos grandes grupos, sistemas multiprocesador y sistemas multicomputador o distribuidos. Los primeros se caracterizan por tener un sistema de memoria compartida, donde todas las unidades de cálculo tienen acceso al mismo espacio de datos, los segundos, en cambio poseen múltiples unidades de cálculo cada una con su propia memoria y se utiliza una conexión de red para comunicarlas. La Figura 1.1 muestra una comparativa entre los dos grupos principales de arquitecturas MIMD, donde P_i representa al procesador i-ésimo y M_i a la unidad de memoria i-ésima. *[VEI 07]*

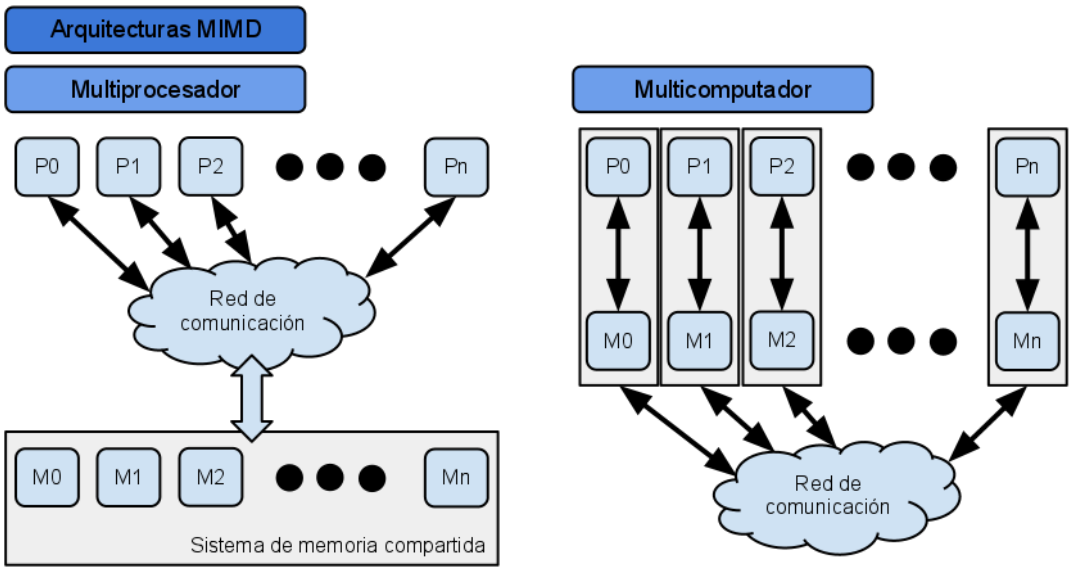

Figura 1.1. Arquitecturas MIMD

La tendencia actual se decanta por los sistemas híbridos, combinando la potencia de

los sistemas multiprocesador con la relación coste-capacidad de cómputo de los sistemas multicomputador.

1.1.2. Programación de aplicaciones distribuidas

Los programas paralelos necesitan un sistema de comunicación entre los procesos que los forman por dos motivos, para garantizar el acceso concurrente con exclusión mutua a recursos compartidos y para intercambiar datos.

Los sistemas distribuidos basan esta comunicación entre procesos en el paso de mensajes, que consiste en la encapsulación de aquellos datos a intercambiar en mensajes que se enviarán a través de la red de comunicación. De esta forma se pueden identificar tres partes esenciales en una comunicación basada en paso de mensajes, el proceso emisor, el proceso receptor y el mensaje. En la Figura 1.2 se muestra un ejemplo de paso de mensajes entre tres procesos A, B y C, donde A divide un vector de datos en dos y envía las partes a los procesos B y C que las reciben, efectúan operaciones sobre ellas y las devuelven al proceso A, que esta bloqueado hasta que las recibe.

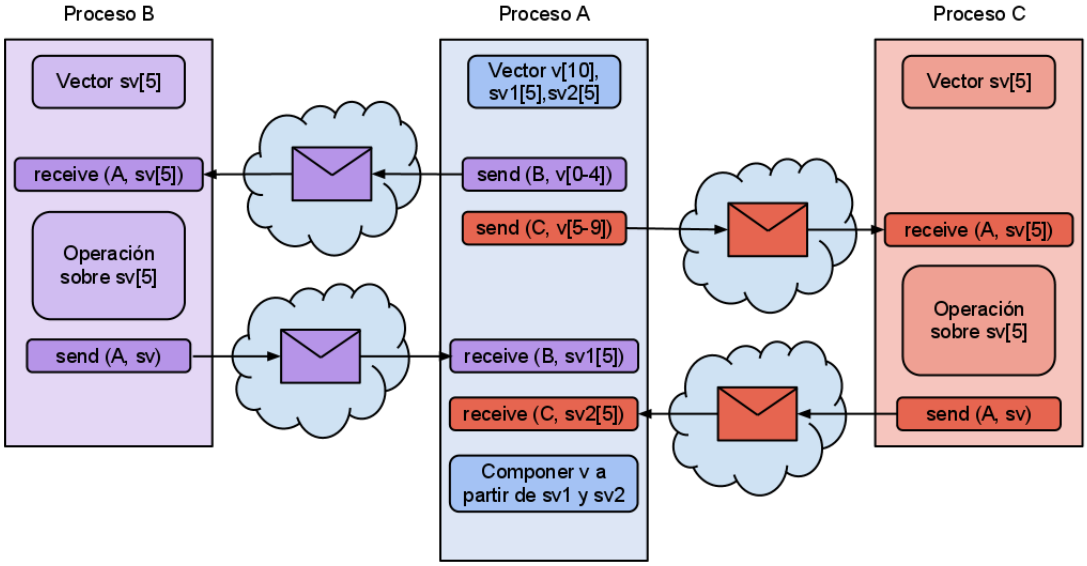

Figura 1.2. Paso de mensajes para distribución de datos

MPI [1] (Message Passing Interface, en inglés), es un protocolo de comunicaciones que se ha convertido en estándar *de facto* para aplicaciones paralelas basadas en paso de mensajes. Las implementaciones del estándar consisten en grupos de funciones (API) que abstraen las capas inferiores de la comunicación de cara al programador.

1.1.3. Análisis de rendimiento y modificación de aplicaciones paralelas

El principal objetivo de las aplicaciones paralelas es sacar provecho de los sistemas paralelos para resolver problemas complejos. Pero esta meta es, en algunos casos, difícil de conseguir puesto que el rendimiento de una aplicación paralela depende de diferentes factores, algunos de ellos arbitrarios, como los datos de entrada, el diseño del algoritmo, el software intermedio, el compilador o la propia habilidad del programador en el momento de pararelizar la solución.

El análisis de aplicaciones paralelas se basa en un ciclo de monitorización-análisis-modificación. Primero, en la fase de monitorización se inserta instrumentación en la aplicación para obtener datos respecto a su comportamiento, luego, en la fase de análisis se contrastan con modelos teóricos y se determinan que cambios hacer en la fase de modificación para acercarse al rendimiento máximo según dichos modelos.

Existen varias herramientas para simplificar el proceso de mejora de aplicaciones paralelas, capaces de automatizar el ciclo e, incluso, de llevarlo a cabo durante la ejecución de la aplicación. A continuación se expondrán diferentes aproximaciones basándose en el uso de este tipo de herramientas. *[CAY 07]*

Análisis de rendimiento clásico

El análisis de rendimiento clásico consiste en un análisis manual de los datos extraídos de la aplicación una vez acabada su ejecución (post-mortem) y en la modificación del código fuente de la misma en función de las conclusiones extraídas del mismo.

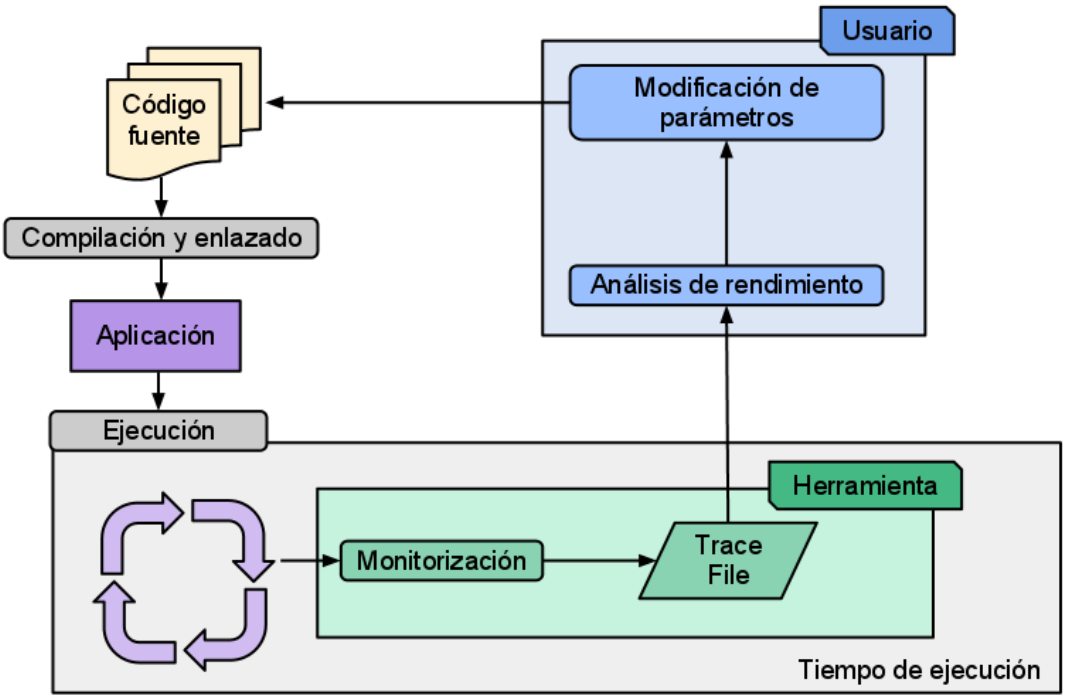

Figura 1.3. Enfoque de análisis de rendimiento clásico

Como se puede observar en la Figura 1.3, en tiempo de ejecución se toman datos de la aplicación objetivo mediante una herramienta de monitorización, que puede instrumentar la aplicación de forma estática o dinámica; con los datos obtenidos se crea un archivo de registro (trace file) que será analizado por el usuario manualmente una vez acabada la aplicación, una vez hecho esto, el usuario introducirá los cambios oportunos en el código fuente, que tendrá que ser recompilado y reenlazado para que se hagan efectivos.

Análisis de rendimiento automático

Un primer paso adelante a partir del sistema clásico consiste en la automatización del proceso de análisis. Esto tiene especial valor puesto que el análisis de un archivo de registro puede llegar a ser un trabajo largo y tedioso, ya que el volumen de información que contienen suele ser muy grande como para ser estudiados manualmente de forma efectiva en el tiempo.

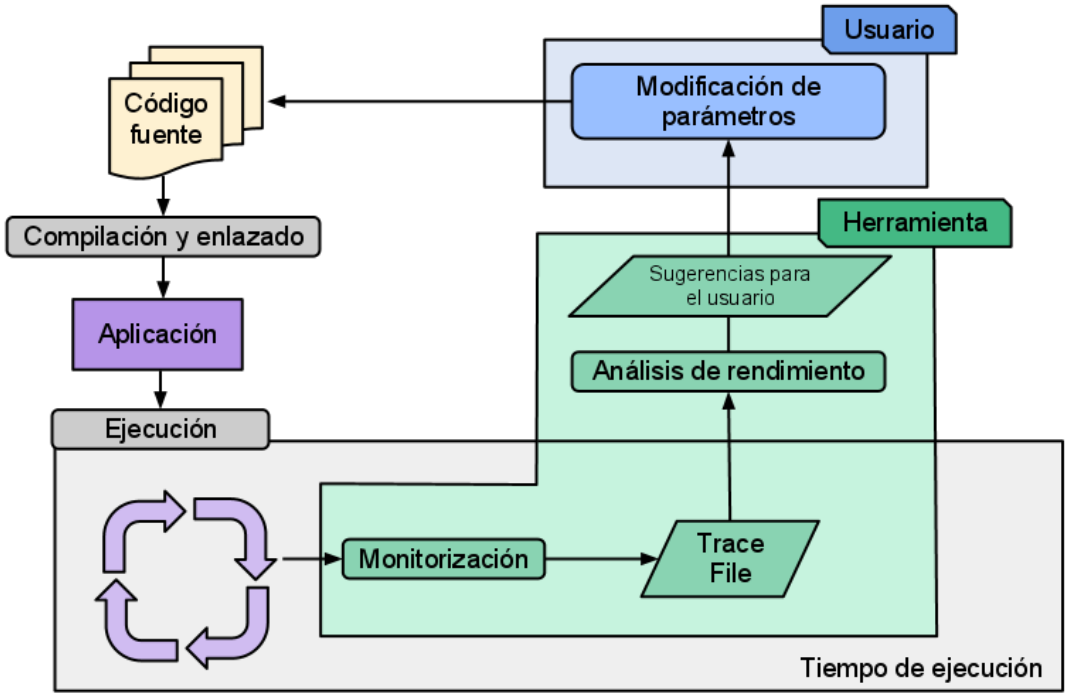

Figura 1.4. Enfoque de análisis de rendimiento automático

Como se puede observar en la Figura 1.4, el ciclo es similar al de la aproximación clásica, pero en este caso el análisis es llevado a cabo por una herramienta que deja al usuario la modificación del código fuente en función de los resultados del análisis como única tarea.

Análisis de rendimiento dinámico

El análisis de rendimiento dinámico busca resolver el mayor inconveniente del análisis automático propuesto anteriormente, el uso de archivos de registro, que implican un tiempo de análisis grande debido a su extensión.

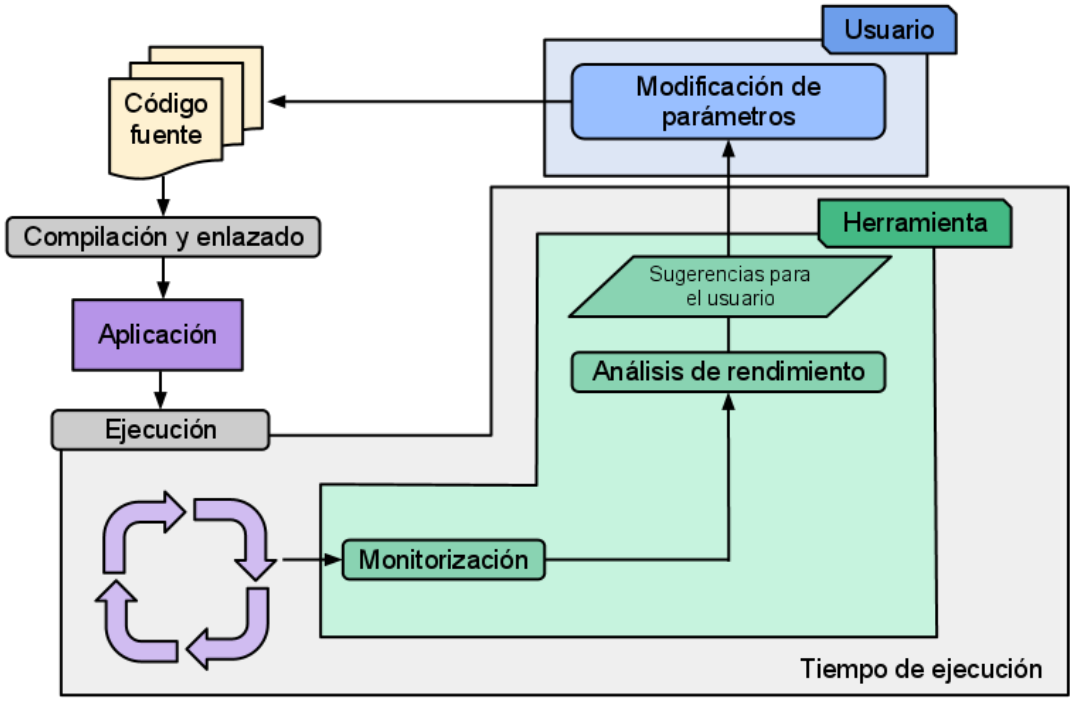

Figura 1.5. Enfoque de análisis de rendimiento dinámico

Al introducir el análisis en el ciclo de ejecución, como se muestra en la Figura 1.5, se obtienen resultados más fiables de forma más rápida puesto que se analiza la información conforme va llegando y no es necesario recorrer un archivo de registro (que suele ser grande).

Para mejorar el proceso en general, se suelen utilizar herramientas de monitorización dinámicas capaces de modificar los parámetros a observar en tiempo de ejecución y que de esta forma el análisis pueda centrarse en aquellos más relevantes en cada momento.

Modificación dinámica

La modificación dinámica consiste en la introducción de los cambios propuestos en la fase de análisis durante la ejecución del programa. Este hecho exime al usuario de intervenir en el ciclo de ninguna manera, pues es una herramienta quien decide y aplica los cambios a realizar.

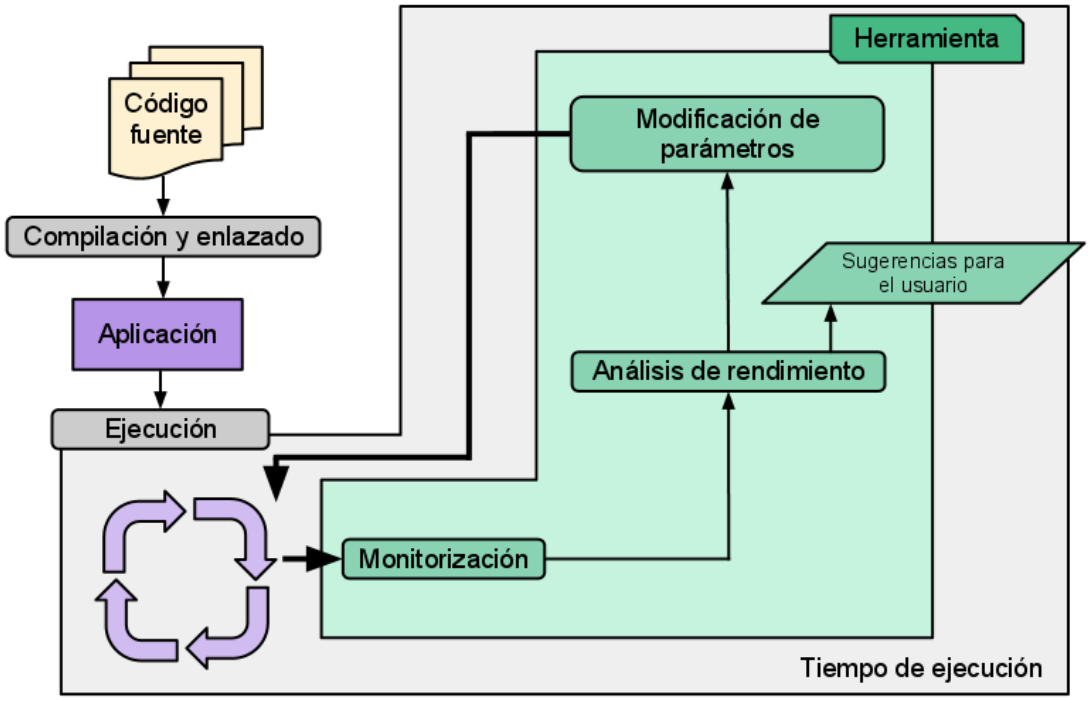

Figura 1.6. Enfoque de análisis y modificación dinámicos

En la Figura 1.6 se muestra el ciclo completo de monitorización-análisis-modificación dentro del tiempo de ejecución, alternativamente también se crean sugerencias para el usuario para el desarrollo futuro.

Esta aproximación presenta multitud de ventajas, la principal que todo el proceso se lleva a cabo durante la ejecución del programa y que por lo tanto no es necesario recompilarlo y reenlazarlo para mejorar su rendimiento.

Además, como se comentaba anteriormente, elimina la participación del usuario dentro del ciclo, hecho particularmente importante puesto que esta clase de aplicaciones están destinadas a usuarios no expertos en la materia (científicos de diversas ramas que buscan beneficio en la computación de altas prestaciones, principalmente).

Por otra parte, además de lo ya expuesto, esta aproximación permite unos resultados mucho más precisos y consistentes que cualquier otra puesto que el análisis es llevado a cabo teniendo en cuenta los datos sobre el rendimiento de la aplicación teniendo en cuenta los parámetros específicos para esa ejecución (hecho que cobra especial

relevancia en aplicaciones que varían mucho sus parámetros de ejecución en función de las condiciones, para las cuales esta aproximación es la única factible).

Dyninst

Dyninst [2] (Dynamic Instrumentation) es una API desarrollada por la Universidad de Wisconsin-Madison y por la Universidad de Maryland cuyo objetivo es generar código en tiempo de ejecución. Para esto Dyninst proporciona una librería de C++ que ofrece primitivas para:

- Crear código e insertarlo en procesos.
- Acceder a código y estructuras de datos de un proceso.
- Remover código insertado previamente.

De esta forma, Dyninst se vuelve la piedra angular para los paradigmas dinámicos presentados en los apartados anteriores puesto que es la base para lograr la monitorización y la modificación de aplicaciones en tiempo de ejecución.

La librería de C++ utiliza una serie de abstracciones para referirse a los diferentes elementos participes en el proceso de modificación automática:

- Mutatee: aplicación a ser instrumentada.
- Mutator: programa que controla y modifica al programa mutado mediante Dyninst.
- Point: punto de la aplicación donde se inserta el nuevo código.
- Snippet: trozo de código ejecutable que se inserta en la aplicación.
- Image: representación del programa en disco.

1.1.4. MATE como aplicación

MATE (Monitoring, Analysis and Tuning Environment) *[MOR 04]* es una herramienta que implementa modificación automática y dinámica de aplicaciones paralelas, en síntesis permite una mejora del rendimiento de aplicaciones paralelas en tiempo de ejecución y sin intervención del usuario.

El ciclo principal de ejecución del entorno consiste en tres fases, monitorización de la aplicación objetivo, análisis de sus parámetros de ejecución y modificación de los mismos. La Figura 1.7 muestra este ciclo y como se hace en tiempo de ejecución.

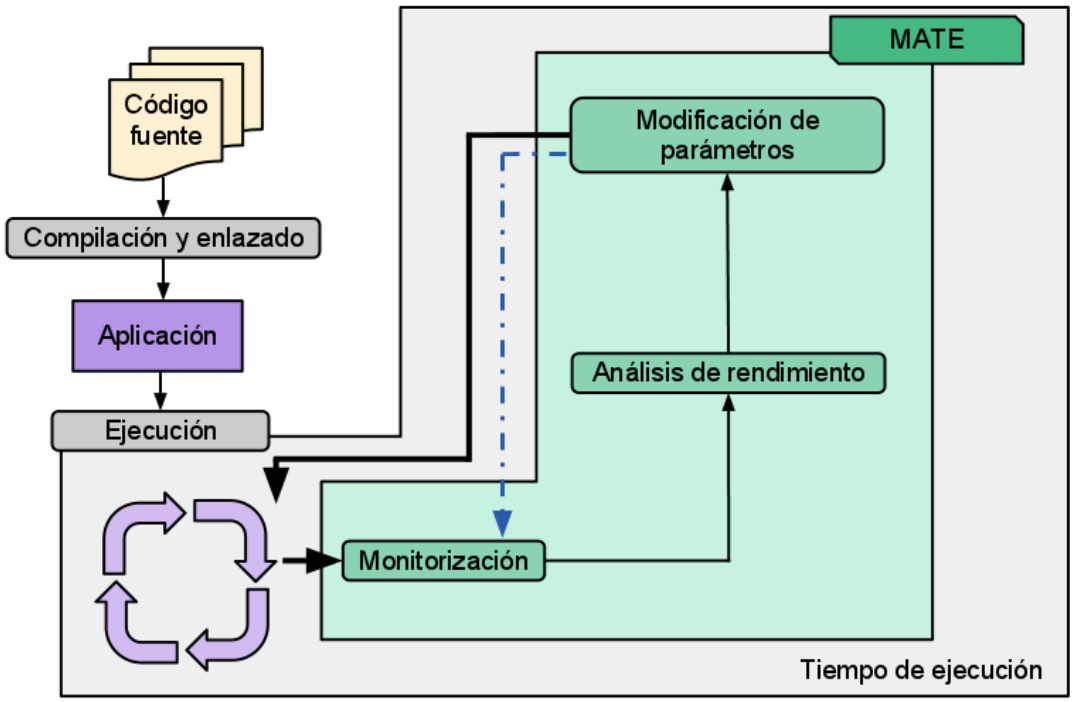

Figura 1.7. Ciclo de ejecución de MATE

La fase de monitorización consiste en instrumentar de forma dinámica y automática (mediante Dyninst) la aplicación objetivo para conseguir información sobre su comportamiento.

La fase de análisis consiste en contrastar los resultados de la monitorización con los proporcionados por modelos teóricos de comportamiento para buscar cuellos de botella, detectar sus causas y determinar soluciones para ellos.

La fase de modificación consiste en la introducción de los cambios determinados en la fase de análisis en la aplicación objetivo utilizando Dyninst. Además se pueden hacer cambios sobre los parámetros a monitorizar de forma que estos sean los más relevantes para el análisis.

MATE implementa esta funcionalidad mediante tres módulos DMLib, AC y Analyzer:

- DMLib (Dynamic Monitoring Library): librería compartida que se carga en la aplicación objetivo y que permite, por una parte, la inserción de instrumentación para recoger datos sobre su ejecución y, por la otra, la modificación de parámetros para mejorarla.
- AC (Application Controller): proceso que se encarga de controlar la ejecución de cada una de las tareas que conforman la aplicación objetivo en cada uno de los nodos.
- Analyzer: proceso central que lleva a cabo el análisis del rendimiento de la aplicación objetivo y decide que parámetros monitorizar y cuales modificar para mejorarlo.

La Figura 1.8 muestra un ejemplo de ejecución de MATE en un cluster de tres nodos, en azul se muestran los componentes de MATE y en rojo los de la aplicación a modificar.

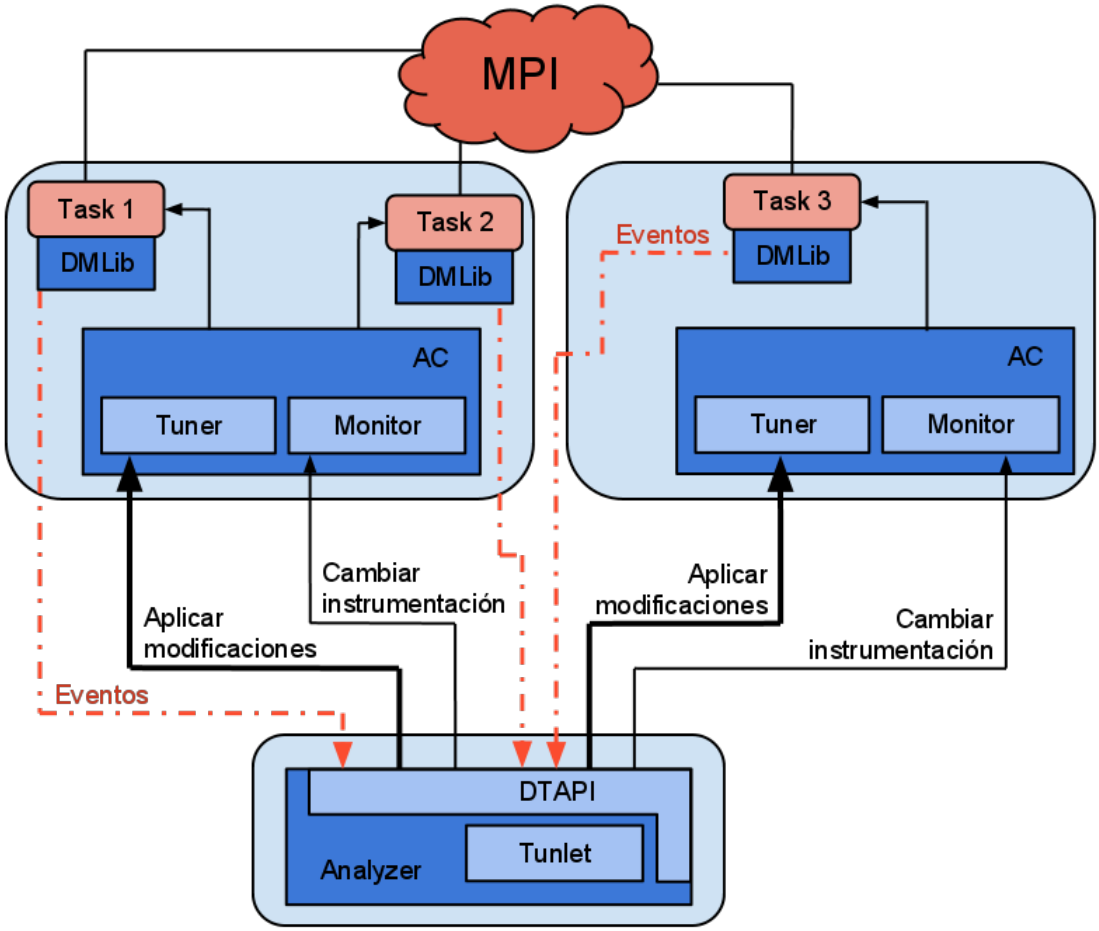

Figura 1.8. Desplegamiento de MATE en 3 nodos

1.1.5. Nuestro proyecto dentro del proyecto MATE

Nuestro proyecto supone un paso adelante en cuestiones de calidad de software y pretende dotar al proyecto MATE de una base de desarrollo sólida de cara a futuras lineas de trabajo. Para ello se pretende hacer frente a la problemática desde tres perspectivas: la creación de una metodología de desarrollo (y su aplicación sobre el proyecto existente), la implantación de un entorno de desarrollo de soporte y el desarrollo de nuevas características para favorecer la portabilidad y la usabilidad, entre otros aspectos.

1.2. Alcance

El marco en el que se encuentra este proyecto es el de la calidad de software y sus objetivos (incluido el desarrollo de nuevas características) están completamente enfocados a ésta.

Por lo tanto, quedan excluidos del alcance de este proyecto todos aquellos aspectos relacionados directamente con la computación de altas prestaciones en un sentido estricto (redes de computadores, sintonización de procesos, algoritmia de aplicaciones distribuidas, etc.).

1.3- Objetivos

Debido a su envergadura, los objetivos de este proyecto se dividen entre los diferentes miembros del equipo de desarrollo. La Tabla 1.1 lista los objetivos del proyecto, los clasifica por prioridad y muestra su miembro asignado. Los objetivos destacados en gris son aquellos que se trabajaran en esta memoria.

	Objetivo	Prioridad	Miembro Asignado*
1	Crear especificaciones del entorno de desarrollo	Prioritario	Grupo
2	Implantar entorno de desarrollo	Crítico	
2.1	Herramienta de colaboración	Crítico	Rodrigo Echeverría, Antonio Pimenta
2.2	Herramienta de control de versiones	Crítico	Antonio Pimenta
2.3	Herramienta de construcción	Crítico	Noel De Martin
3	Construir la metodología de desarrollo.	Crítico	Grupo
3.1	Guía de estilo de documentación.	Crítico	Rodrigo Echeverría
3.2	Guía de estilo de codificación.	Crítico	Noel De Martin
3.3	Guía de estilo de documentación de código.	Prioritario	Rodrigo Echeverría

3.4	Guía de estilo de desplegamiento.	Prioritario	Antonio Pimenta
4	Aplicar la metodología y especificaciones a MATE.	Crítico	
4.1	Aplicación sobre las clases comunes (Common)	Crítico	Noel De Martin
4.2	Aplicación sobre el módulo DMLib.	Crítico	Noel De Martin
4.3	Aplicación sobre el módulo AC.	Crítico	Antonio Pimenta
4.4	Aplicación sobre el módulo Analyzer.	Crítico	Rodrigo Echeverría
5	Desarrollo de nuevas características	Secundario	
5.1	Crear un instalador para cualquier versión de Linux.	Secundario	Rodrigo Echeverría, Antonio Pimenta
5.2	Crear un lector de configuraciones flexible.	Secundario	Noel De Martin
5.3	Crear un sistema de cerrado de MATE.	Secundario	Rodrigo Echeverría, Antonio Pimenta
6	Crear documentación de MATE para futuros colaboradores y usuarios.	Prioritario	Grupo

Tabla 1.1. Objetivos generales del proyecto

**Puesto que en algunos casos la división de trabajo se hizo a posteriori para balancear la carga, en los casos en que existe más de un miembro asignado se detallará en capítulos posteriores.*

Los filas de la Tabla 1.1 resaltadas indican que objetivos son perseguidos en la presente memoria.

1.4- Estructura documento

El trabajo presentado en esta memoria se organiza en siete capítulos, el primero de los cuales es esta misma introducción.

El segundo capítulo, Plan de proyecto y viabilidad, presenta la planificación para conseguir los objetivos del proyecto: división de tareas, calendario y recursos disponibles y, a partir de estos datos, se hace un análisis de viabilidad tanto técnica como económica.

El tercer capítulo, Calidad de software, presenta este concepto que actúa como motivación y eje principal de cara al desarrollo del proyecto. Además expone una serie de modelos para evaluar la calidad de software y presenta las tres lineas de proyecto, desarrolladas en cada uno de los tres capítulos posteriores, que buscan satisfacer dichos modelos.

El cuarto capítulo, Definición e implantación de la metodología de desarrollo, se muestra una perspectiva general sobre los principales elementos que la conforman, las guías de estilo, y se detalla la contribución personal respecto a la confección de éstas. Una vez explicados todos los aspectos de la metodología, se muestra el proceso de implantación de ésta sobre el código actual de MATE.

El quinto capítulo, Implantación del entorno de desarrollo, describe como se llevó a cabo la instalación de un sistema de desarrollo para el proyecto MATE. Primero se detallan las características buscadas para el sistema y se presenta una visión de conjunto de las herramientas escogidas para satisfacerlas y, luego, se especifica la división de trabajo entre los miembros del grupo.

Una vez definido el entorno global, se detallan las características de la herramienta asignada y se muestra el proceso de implantación: instalación, confección del manual de instalación y configuración, integración y automatización del proceso.

El sexto capítulo, Desarrollo de nuevas características, muestra el proceso de creación de dos módulos nuevos para MATE, el módulo instalador y el módulo de cerrado, para cada uno de ellos se describe el proceso mediante el modelo en cascada: análisis de requerimientos, diseño, codificación y prueba.

Finalmente, el séptimo capítulo, Conclusión, resume el trabajo mostrado haciendo referencia a la consecución de objetivos y a las posibles lineas de trabajo futuras.

2. Plan de proyecto y viabilidad

En este capítulo se presenta el estado actual del proyecto MATE, sus perspectivas de futuro y los requisitos que tiene nuestro proyecto para él. Por otra parte, se detalla la planificación necesaria para cumplir con sus objetivos: división de tareas, calendario y recursos disponibles. Finalmente se hacen análisis de viabilidad técnica y económica. Está es una versión reducida del plan de proyecto, se puede consultar en su totalidad en el anexo 6.

2.1. Situación actual

2.1.1. MATE como proyecto

MATE nace dentro de la tesis doctoral de Anna Morajko referida a la modificación dinámica de aplicaciones distribuidas. Su objetivo era crear una aplicación que pudiera llevar a la práctica el concepto de modificación automática y dinámica de aplicaciones paralelas y de esa forma observar si éste es realmente posible sin representar un grado de intrusión (carga de la red, carga de la máquina) elevado.

Las características del sistema en su planteamiento eran:

- Monitorización, análisis y modificación en tiempo de ejecución.
- Independencia de la aplicación objetivo (ésta no debe ser recompilada ni reenlazada para funcionar con MATE).
- Modificación segura de la aplicación objetivo.
- Baja intrusión, puesto que el objetivo era mejorar el rendimiento de la aplicación, la implementación de MATE ha de minimizar su propio nivel de intrusión.

- Análisis ligero, para poder tomar decisiones de modificación dentro de intervalos de tiempo donde son relevantes.
- Sistema de modificación abierto y extensible para que desarrolladores puedan introducir nuevas técnicas de modificación.
- Facilidad de uso para usuarios finales, puesto que el perfil de estos no es típicamente el de desarrollador de software.

Posteriormente el proyecto MATE creció y derivó en diferentes proyectos destinados a ampliar su funcionalidad, mejorar su rendimiento, aumentar su facilidad de uso, etc. En esta linea se encuentran los trabajos de Paola Cayme *[CAY 07]* referido a la automatización en la creación de las técnicas de sintonización y su incorporación a MATE y de Andrea Martínez sobre la experimentación con nuevos modelos de rendimiento y escalabilidad.

El trabajo actual y futuro se centra en adaptar MATE a los sistemas de gran escala. Eso tiene influencia en la representación de las técnicas de sintonización ya que ya no habrá un analizador central, sino el análisis se hará a varios niveles para poder abarcar una aplicación que se ejecuta en miles de nodos

2.1.2. Lógica del sistema

Como la base de la lógica de MATE ya ha sido explicada en la introducción (sección 1.1.4), este apartado se dedicará a profundizar en la lógica de Analyzer, que es el módulo sobre el que trabaja la presente memoria.

Analyzer se encarga de hacer el análisis de rendimiento de la aplicación, automáticamente detecta problemas de rendimiento en tiempo de ejecución y pide los cambios necesarios para mejorarlo. El análisis se basa por una parte, por conocimiento sobre la aplicación que viene desde fuera y, por la otra por la monitorización del rendimiento basada en el trazado de eventos. *[MOR04]*

Para conseguir este objetivo Analyzer se basa en dos partes, la API de modificación

dinámica (DTAPI) y los tunlets. DTAPI actúa como una interfaz para la monitorización y la modificación y los tunlets aportan la lógica necesaria para el análisis. La implementación de DTAPI se basa en tres módulos: Application Manager, Comunicator y el Event Collector. La Figura 2.1 aporta una visión de conjunto del sistema.

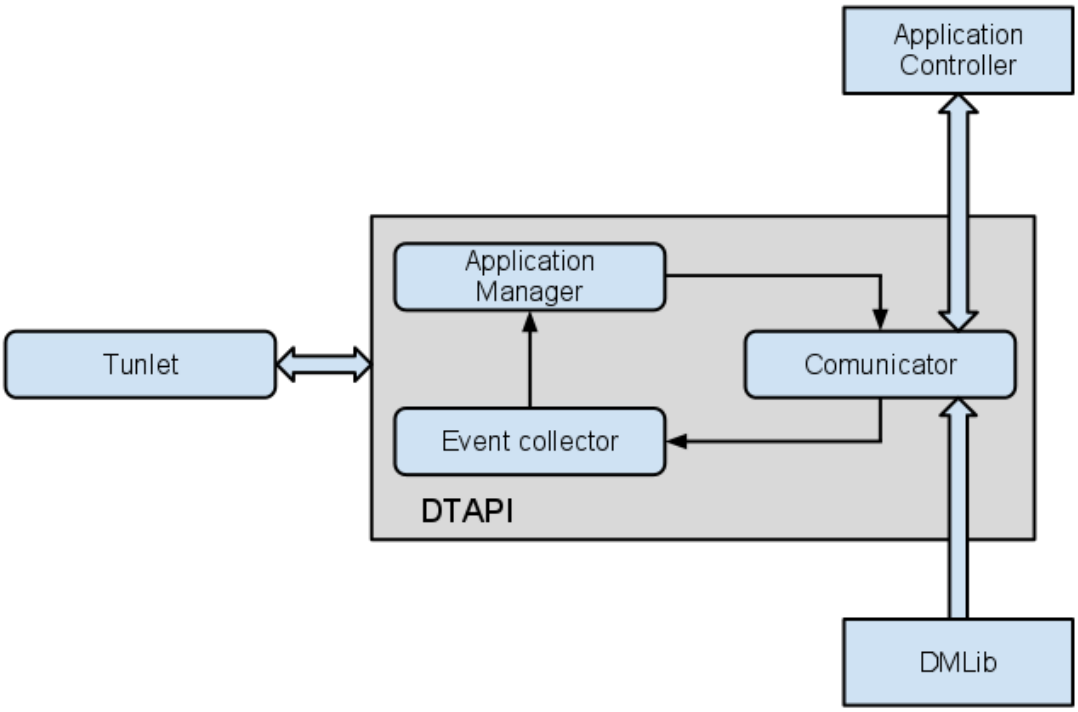

Figura 2.1. Ciclo de ejecución de Analyzer

El ciclo de ejecución de DTAPI consiste en recolectar información sobre la ejecución de la aplicación objetivo desde los *DMLib's* cargados en cada nodo. Esta información es preprocesada en el *Event Collector* y es pasada a los tunlets para que la analicen. Una vez decididas las medidas a tomar el *Application Manager* utiliza el *Comunicator* para delegar en los AC (*Application Constroller*) la responsabilidad de hacer cambios.

2.2. Requisitos funcionales y no funcionales

Debido, principalmente, a que este proyecto no es un proyecto de desarrollo puro, ya que implica tres lineas de proyecto diferentes (definición e implantación de una metodología de desarrollo, implantación de un entorno de desarrollo y creación de nuevas características) es difícil concretar requisitos funcionales y no funcionales (ya que estos conceptos están referidos a qué hace un sistema y bajo que condiciones).

En un sentido amplio, el objetivo de las tres lineas de proyecto es apoyar en aspectos no funcionales de MATE, puesto que éste tal, y como, se indica en la sección anterior, es un software totalmente funcional. Los requisitos no funcionales que se cubren abarcan aspectos como la mantenibilidad, la usabilidad o la portabilidad y son detallados en el capítulo 3, referido a aspectos de la calidad del software.

2.3. Alternativas y selección de la solución

Como se comentaba en la sección anterior, el hecho de tener tres lineas de proyecto independientes también afecta al conjunto de alternativas. A diferencia de un proyecto de desarrollo donde se estudiarían alternativas respecto al sistema a desarrollar, en este proyecto se deben estudiar alternativas respecto al modelo base para la metodología, cada uno de los elementos del entorno de desarrollo y a la forma de implementar las nuevas características.

Para facilitar la lectura y favorecer la cohesión de la memoria, se procederá, por lo tanto, a delegar en cada capítulo (referidos a cada linea de proyecto) la responsabilidad de exponer alternativas y de escoger una solución entre ellas.

2.4. Planificación

2.4.1. WBS (Work Breakdown Structure)

El conjunto de tareas del proyecto se dividirá en 5 ramas, una para cada una de las tres lineas de proyecto más una de estudio previo y otra de cierre. La Figura 2.2 muestra la jerarquía de tareas completa así como los artefactos que generan.

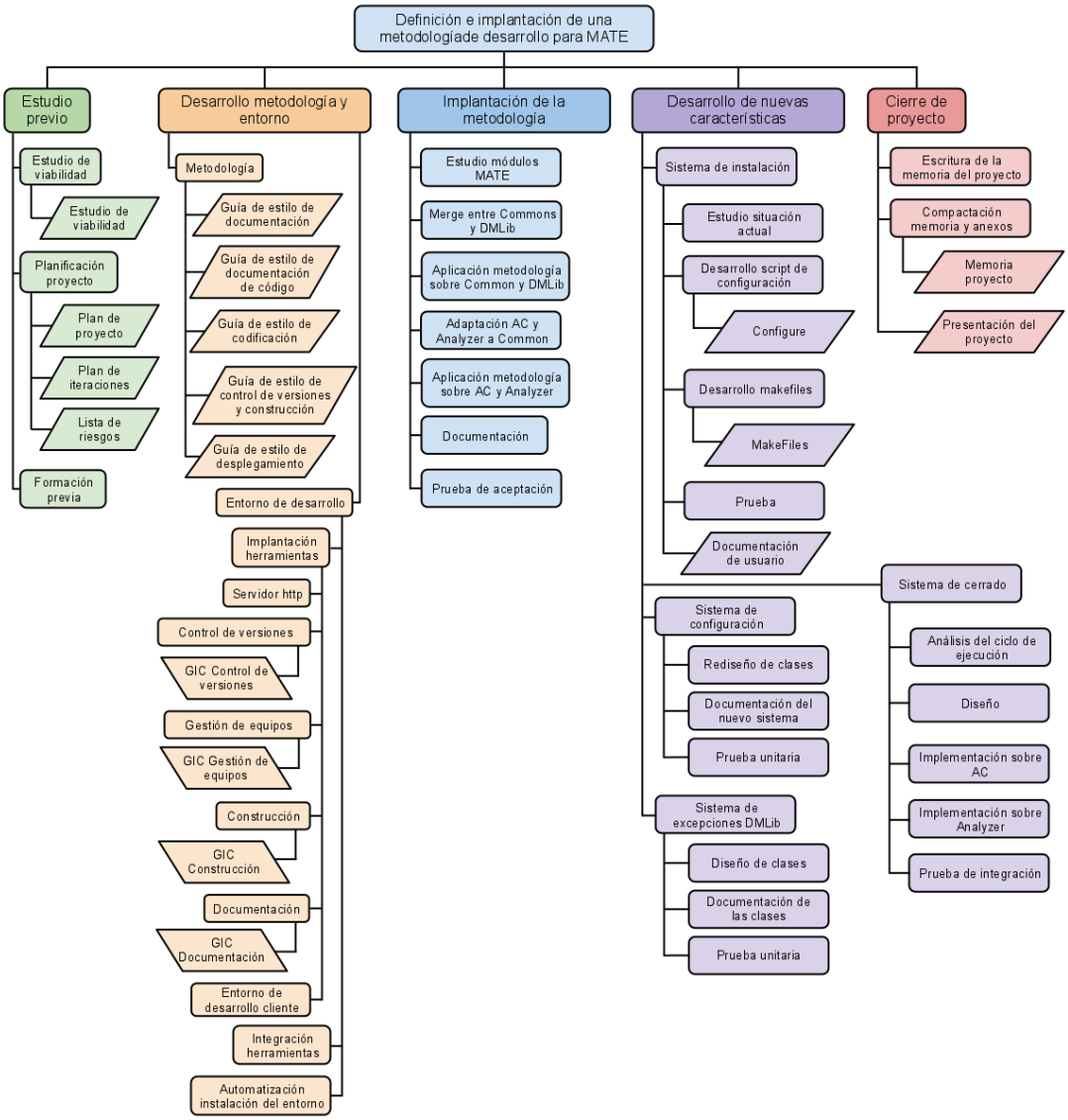

Figura 2.2. Diagrama WBS

2.4.2. Fases y actividades del proyecto

Fase	Actividad	Descripción
Estudio previo	Estudio de viabilidad	Estudio para analizar las posibilidades y mejores alternativas para realizar el proyecto y si estás son posibles con los recursos disponibles en el tiempo requerido.
	Planificación del proyecto	Análisis sobre las tareas que compondrán el proyecto, su calendario, los recursos necesarios para ejecutarlas y los riesgos que comportan a la consecución del proyecto.
	Formación previa	Estudio sobre los temas relacionados con paso de mensajes (MPI) y sintnización de procesos (Dyninst)
Desarrollo metodología y entorno	Desarrollo guías de estilo	Creación de los documentos que conforman la base de la metodología a implantar: guías de estilo de documentación, codificación, construcción, control de versiones, etc.
	Implantación entorno	Selección de las diferentes herramientas que forman el entorno de desarrollo, implantación e integración de las mismas.
Implantación metodología	Estudio módulos MATE	Estudio sobre el código en su versión original de cada módulo de MATE.
	Combinación DMLib y Commons	Eliminación de las clases redundantes entre Commons y DMLib y refactorización de AC y Analyzer en consecuencia.
	Documentación y refactorizacion	Documentación sobre el código de cada módulo y refactorización derivada de las guías de estilo.
	Prueba unitaria	Prueba unitaria de las clases que componen cada módulo.
	Documentación	Extracción y compilación de la documentación sobre código.
	Prueba de aceptación	Prueba del sistema completo para detectar posibles errores en la refactorización.
Desarrollo de nuevas características	Sistema de instalación	Desarrollo de un sistema de instalación capaz de automatizar la búsqueda de dependencias y la compilación en el mayor grado posible.

	Sistema de configuración	
	Sistema de cerrado	Desarrollo de un sistema capaz de cerrar el entorno de forma centralizada y controlada.
Cierre de proyecto	Escritura y compilación de la memoria	Escritura de la memoria del proyecto y compilación del documento junto con los anexos que lo acompañan.
	Exposición del proyecto	Exposición del proyecto ante un tribunal para su evaluación.

Tabla 2.1. Fases y actividades del proyecto

2.4.3. Recursos del proyecto

Definición de recursos

Recursos humanos

- 3 Programadores-Analistas: Noel De Martín, Rodrigo Echeverría, Toni Pimenta
- 1 Project Manager: Joan Piedrafita

Recursos de infraestructura

- Servidor de proyectos, colaboración y builds virtualizado
  - Equipo: Dell PowerEdge R515
  - Procesador: 2 x AMD Opteron 4122 (4 Cores – 2.2 Ghz – L1 3MB / L2 6MB – 95W TDP)
  - Memoria: 8GB Memory for 2 CPUs, DDR3, 1333MHz (8x1GB Single Ranked UDIMMs)
  - Disco: 2x 250GB, SATA, 3.5-in, 7.2K RPM Hard Drive (Hot Plug)

- 8 Nodos de computo cluster DiskLess
  - Equipo: Dell PowerEdge R610
  - Procesador: 2 x Intel Xeon E5620 (4 Cores – 2,4 Ghz – 12 MB Cache – QPI 5,86 Gb/s)
  - Memoria: 12GB DDR3, 1333MHz ECC (12x1GB)

- SAN
  - Almacenamiento: DELL™ PowerVault™ MD3200i, 6 discos SAS 7.2k rpm 500 GB
  - Red de gestión: 2 x SWITCH ETHERNET DELL PowerConnect 5424

- Otros
  - Sistema de alimentación: SAI 8000VA APC
  - Switch control cluster:SWITCH ETHERNET DELL PowerConnect 5448
  - Switch gestión: SWITCH INFINIBAND SDR DE 8 PUERTOS, 4X, 1U.
  - Switch Red MATE: SWITCH ETHERNET DELL PowerConnect 5424
  - Rack PDU (8 Tomas + Ethernet)
  - Chasis Rack 42U
  - CABLE INFINIBAND 2 METROS CON CONEXION X4
  - Cable de interconexión - RJ-45 (M) - RJ-45 (M) - 2 m - UTP - (CAT 6)

Configuración de la infraestructura

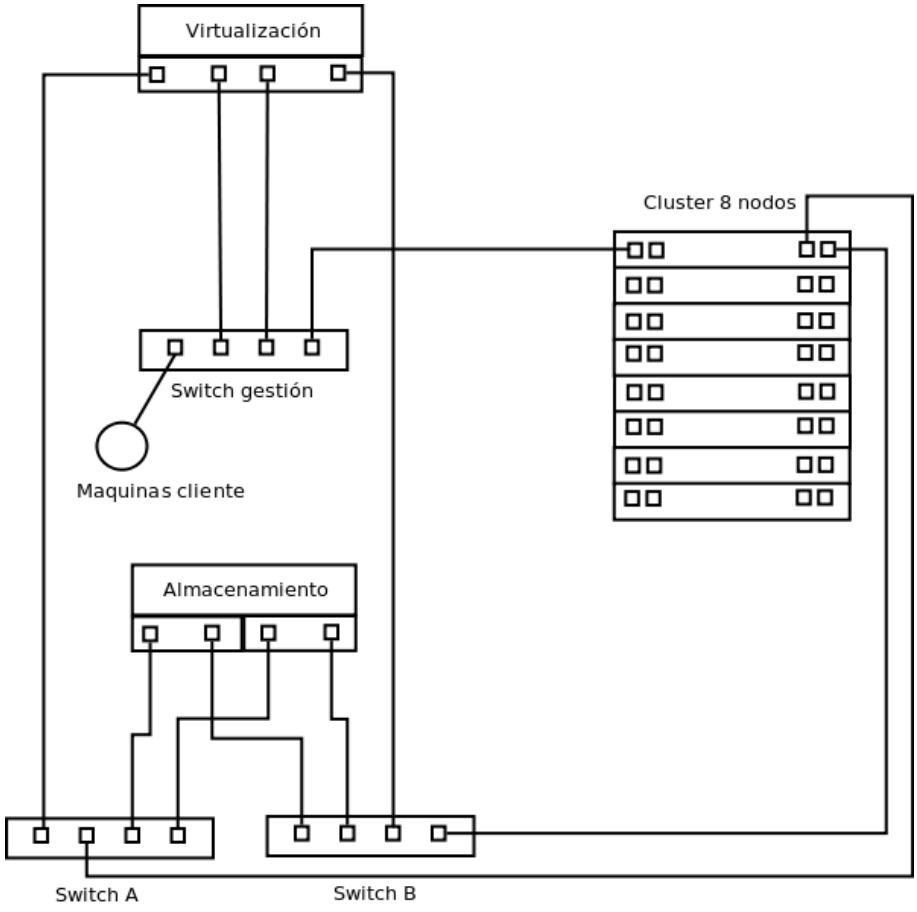

Figura 2.3. Red de desarrollo completa

La Figura 2.3 muestra la red de desarrollo completa, en ella se pueden diferenciar tres partes:

- Virtualización: encargada de alojar máquinas virtuales para prueba.
- Almacenamiento: encargada de almacenar de forma redundada el código y los documentos.
- Cluster de 8 nodos: encargado de proporcionar la infraestructura de prueba de aplicaciones paralelas.

El sistema se detalla posteriormente en la sección 5.2. Especificación del entorno de desarrollo.

Calendario de los recursos

Los recursos humanos se utilizarán durante todo el proyecto, sin embargo el cluster y el almacenamiento solo se utilizarán en la segunda parte del proyecto haciendo las pruebas necesarias con aplicaciones paralelas para comprobar los resultados. El resto de recursos materiales también se utilizarán durante todo el proyecto.

2.4.4. Calendario temporal

La duración total estimada del proyecto es 248 días que, con una dedicación media de 4h/día, implican 992 horas de trabajo a distribuir entre los tres miembros del equipo. Las Figuras 2.4 a 2.10 muestran los diagramas de Gantt para cada fase del proyecto mostrada en el WBS.

Fase 1: Estudio previo

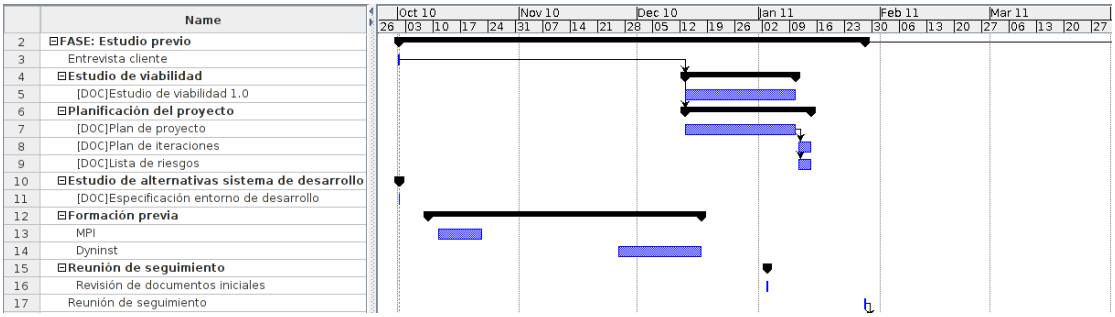

Figura 2.3. Diagrama de Gantt de la fase 1: Estudio previo

Fase 2: Desarrollo metodología y entorno

1ª iteración

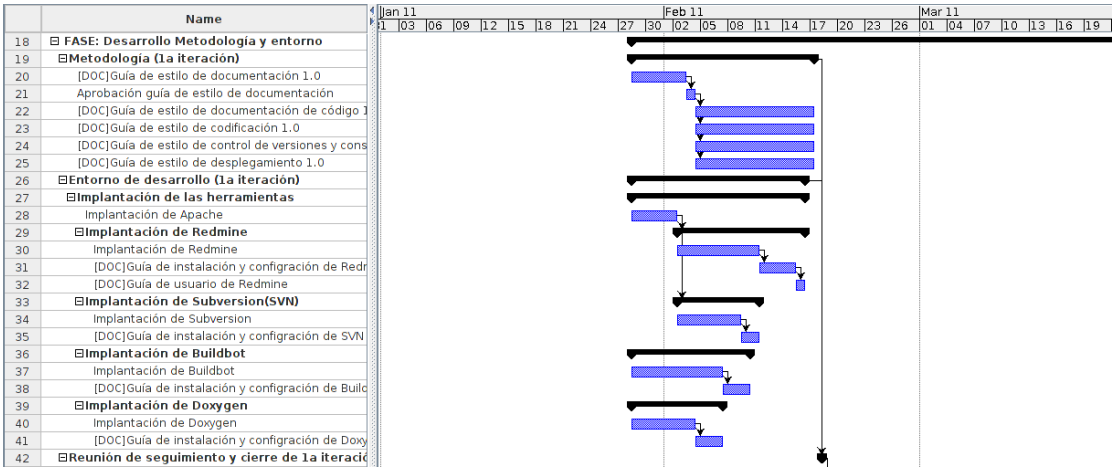

Figura 2.5. Diagrama de Gantt de la fase 2: Desarrollo metodología y entorno (1ª iteración)

2ª iteración

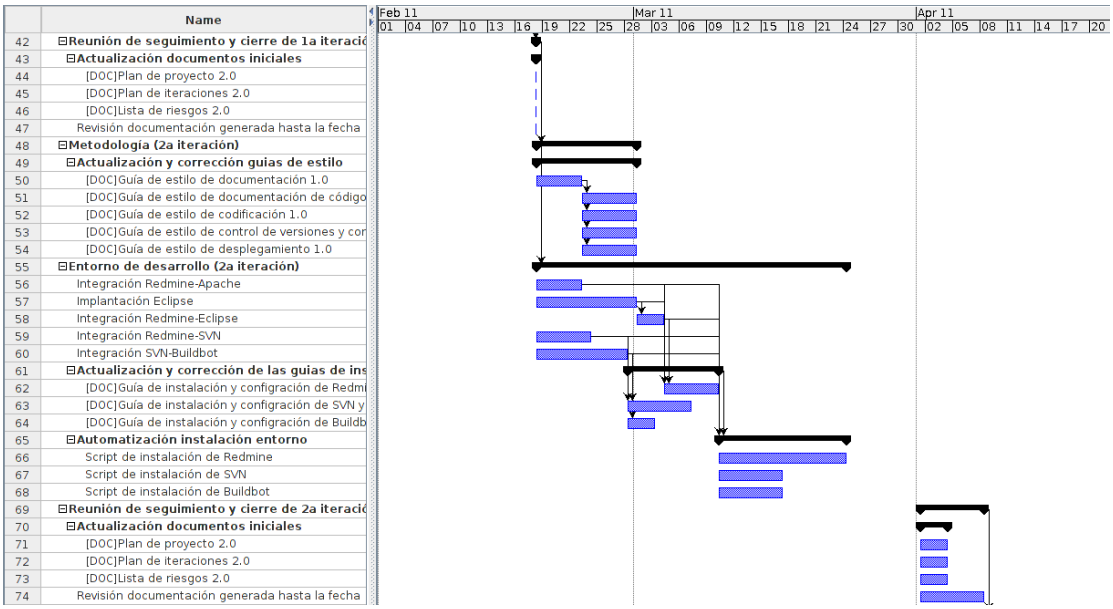

Figura 2.6. Diagrama de Gantt de la fase 2: Desarrollo metodología y entorno (2ª iteración)

Fase 3: Implantación de la metodología

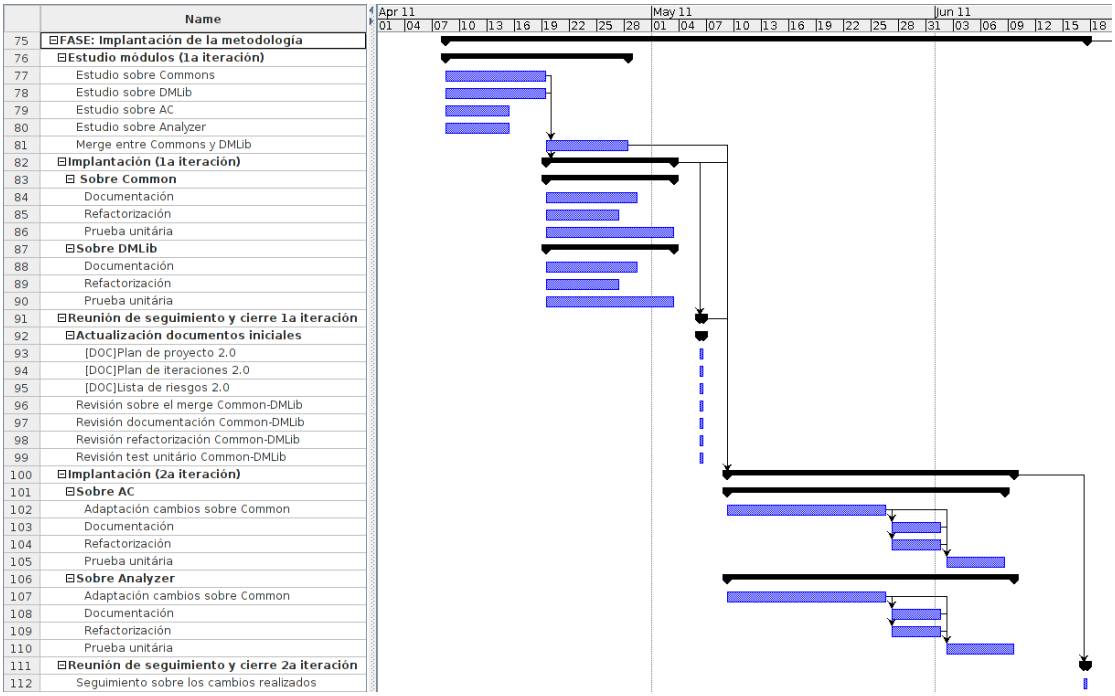

Figura 2.7. Diagrama de Gantt de la fase 3: Implantación de la metodología

Fase 4: Desarrollo de nuevas características

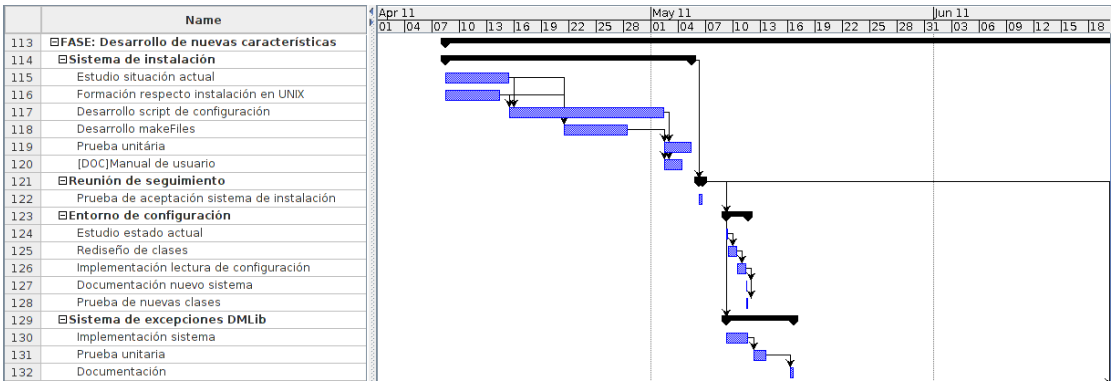

Figura 2.8. Diagrama de Gantt de la fase 4: Desarrollo de nuevas características (1)

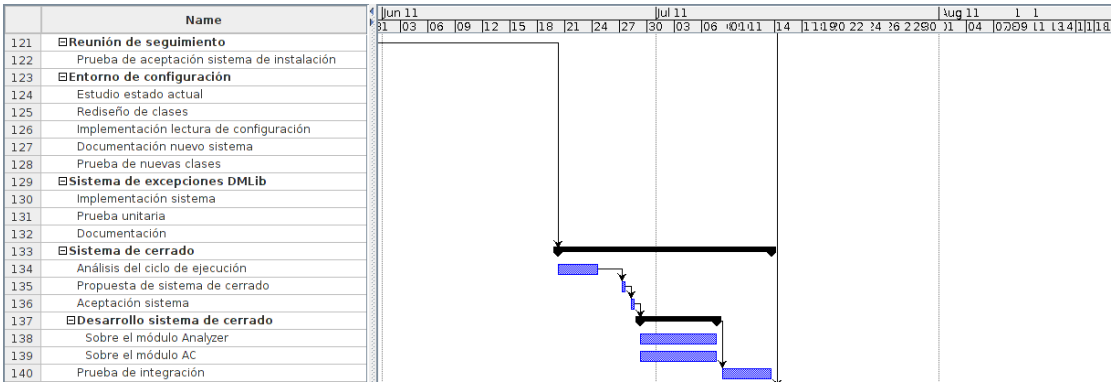

Figura 2.9. Diagrama de Gantt de la fase 4: Desarrollo de nuevas características (2)

Fase 5: Cierre de proyecto

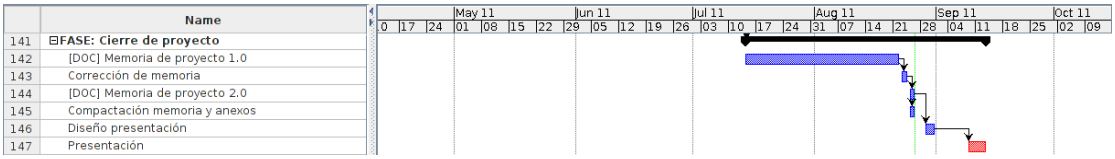

Figura 2.10. Diagrama de Gantt de la fase 5: Cierre de proyecto

2.5. Análisis de la viabilidad técnica

Desde un punto de vista cualitativo, todos los objetivos del proyecto (planteados en la introducción) siguen el criterio SMART (Specific, Measurable, Attainable, Relevant, Timely) y desde un punto de vista cuantitativo, a pesar de ser una gran cantidad están planteados para un equipo de tres técnicos, por todo esto se puede concluir que el proyecto es técnicamente viable.

2.6. Análisis de la viabilidad económica

Para cada apartado se muestra un resumen de coste de los factores que están siendo analizados con una proyección de un año.

2.6.1. Estimación coste de personal

Recurso	Coste
3 Analista-programador	64.500 €
1 Project Manager	28.000 €
Total	92.500 €

2.6.2. Estimación coste de los recursos

Recurso	Coste
Servidor	2.220,76 €
8 nodos cluster	23.902,08 €
SAN	8.173,86 €
Otros	12.917,21 €
Software	0 €
Total	47.213,91 €

2.6.3. Estimación coste de las actividades

Ninguna actividad tiene costes directos.

2.6.4. Estimación de otros costes

Recurso	Coste
Personal de soporte	24.000€
Alquiler local	25.488€
Total	49.488 €

2.6.5 Estimación costes indirectos

Recurso	Coste
Electricidad	5.389,65 €
Consumibles	1.475 €
Telefonía	708 €
Limpieza	4.141,8 €
Mantenimiento	1.062 €
Gestión	2.124 €
Total	14.900,45 €

2.6.6 Resumen y análisis coste beneficio

Cabe matizar en parte este resultado, primero porque los costes de personal nos incluyen y, por lo tanto, se reducen a 0. Por otra parte, los costes de recursos se irán amortizando en los sucesivos proyectos. El resto de costes quedan en negativo, pero el beneficio que comporta el proyecto en cuestiones de investigación y promoción lo compensa.

3. Calidad de software

En este capítulo se presenta el concepto de calidad de software que actúa como motivación y eje principal de cara al desarrollo del proyecto. Primero se hace una perspectiva general sobre la importancia que adquiere dicho concepto en los proyectos software. Luego, se plantean algunos modelos de calidad de software existentes (tanto estándares como desarrollados por empresas). Para finalizar, se presentan los frentes de trabajo del proyecto como medidas para mejorar los aspectos de calidad de software mencionados.

3.1. Calidad en el desarrollo de software

El concepto de calidad aplicado al software se refiere tanto al grado de concordancia que existe entre las especificaciones iniciales (qué se espera que haga) y los resultados finales (qué hace) como a la medida de otros aspectos no relacionados con la funcionalidad como la mantenibilidad, la facilidad de uso, la fiabilidad o la portabilidad.

De esta forma, la garantía de la calidad de software (SQA, del inglés Software Quality Assurance) se vuelve un aspecto clave en el desarrollo del mismo. SQA consiste en un modelo sistemático y planeado de todas las acciones necesarias para asegurar la calidad esperada del producto final, así como la correcta aplicación de estándares y procedimientos adoptados. *[GIE 11]*

La garantía de calidad en el caso del software es una actividad de protección que se debe realizar durante el proceso de desarrollo del mismo (no solamente al final) puesto que el coste de los fallos detectados es mayor cuanto más tarde detecta. Para ilustrar la reducción del coste con la detección anticipada de errores, se pueden considerar una serie de costes relativos que se basan en datos de proyectos de software reales *[IBM81]*. Suponiendo que un error descubierto en la fase de diseño del producto cuesta 1,0 unidad monetaria, este mismo error descubierto antes de realizar el proceso de test costará 6,5 unidades, durante las pruebas 15 unidades, y

después de la entrega entre 60 y 100 unidades. El mismo razonamiento se puede aplicar a otros recursos de un proyecto como pueden ser tiempo o rendimiento.

3.2. Modelos de calidad de software

Existen diferentes modelos de cara a preparar un plan de garantía de calidad de software para un proyecto. Todos ellos listan aspectos a tener en cuenta en el momento de evaluar la calidad del software y proponen una serie de métricas para cuantificarlos.

3.2.1 Modelo McCall

Uno de los primeros modelos existentes, en el que se basan la mayoría de los actuales, es el modelo de McCall que, en un principio, fue creado para las fuerzas aéreas de los Estados Unidos en 1977. Principalmente está enfocado a los desarrolladores del sistema y al proceso de desarrollo. *[FIL 07]*

En este modelo McCall intenta unir la perspectiva de usuarios y desarrolladores. El modelo también es llamado FCM (Factor, Criteria, Metrics) porque presenta los diferentes aspectos de calidad de software en una jerarquía de tres niveles de abstracción:

- Factores: características abstractas de calidad: revisión (habilidad para adoptar cambios), transición (habilidad para adaptarse a otros entornos) y operación (sus características operativas).
- Criterios: descomposición de los factores en elementos concretos.
- Métricas: correspondencia de criterios con atributos del software lo suficientemente específicos como para ser medidos directamente.

La Figura 3.1 presenta la jerarquía de McCall completa. La primera columna corresponde al primer nivel, los factores, cada uno de ellos expande sus criterios correspondientes en horizontal y cada criterio lista sus métricas. Se puede observar que existen métricas que pueden medir criterios diferentes, incluso factores diferentes.

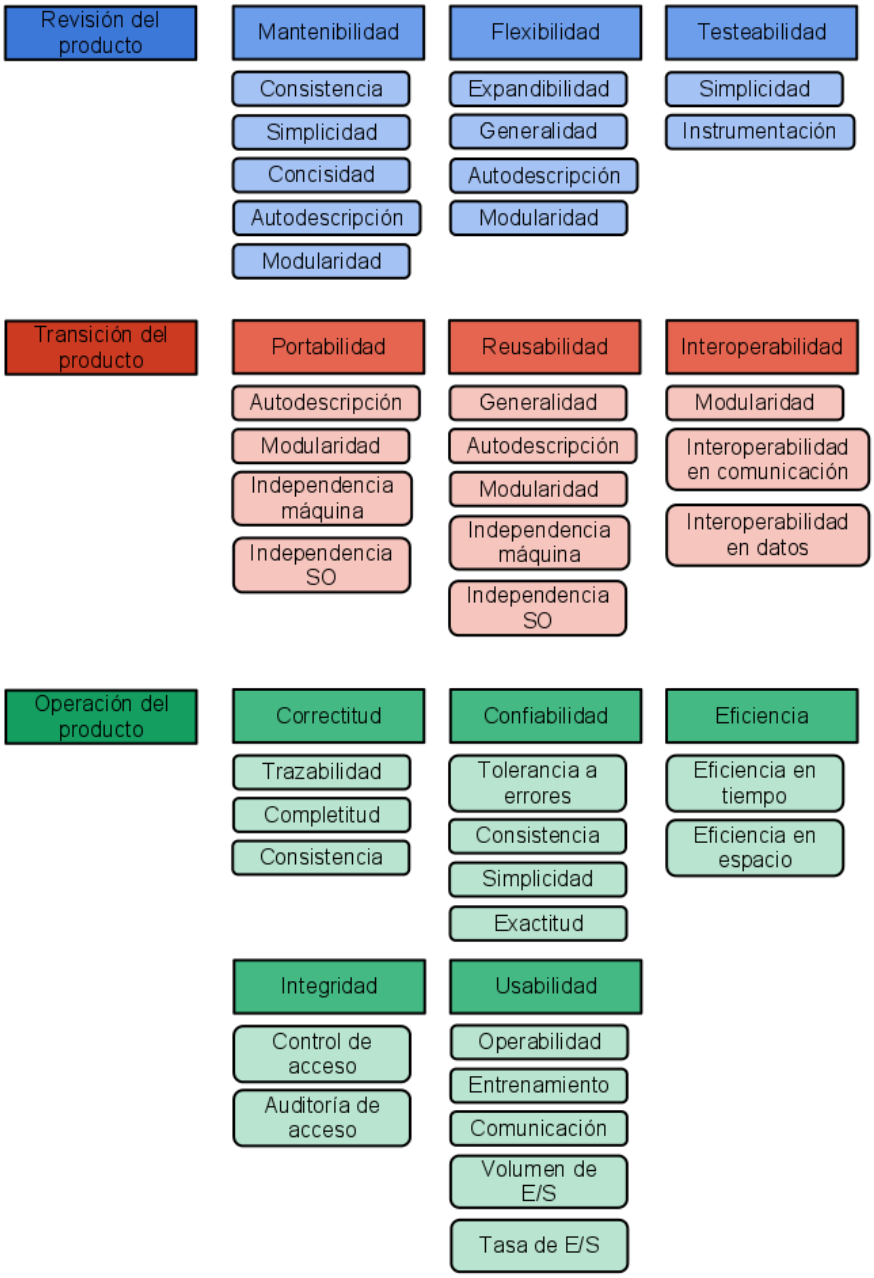

Figura 3.1. Modelo McCall

3.2.2. Modelo FMEA

Otro modelo a mencionar es el FMEA (Failure Mode and Effects Analisis). Como su nombre indica se basa en analizar problemas potenciales, principalmente en una época temprana del ciclo de desarrollo donde es más fácil tomar acciones para solucionarlos. FMEA se utiliza para identificar fallos potenciales en los sistemas, para determinar su efecto sobre la operación del producto, y para identificar acciones correctivas para atenuar las faltas. Podemos encontrar diferentes tipos siguiendo este modelo según su enfoque: Sistema (enfocado a funciones globales del sistema), Diseño (enfocado a componentes y subsistemas), Proceso (enfocado a procesos de fabricación y ensamblamiento), Servicio (enfocado a funciones del servicio) y Software (enfocado a funciones del software).

3.2.3. ISO 9126

La ISO (International Organization for Standardization) ha emitido algunas normas que definen modelos de calidad de software para diferentes contextos de uso. El objetivo era definir una visión unificada del concepto de calidad para poder comparar productos.

Una de ellas es la ISO 9126 que surge en 1992 como una variante del modelo McCall presentado anteriormente (sección 3.2.1). Esta norma se basa en la idea que el foco de calidad cambia durante la vida del producto (no existe una calidad absoluta), en general, el software en sus diferentes etapas ha de satisfacer a los clientes (los resultados han de corresponderse con las especificaciones), a los desarrolladores (el diseño tiene que corresponder con los requisitos, el código con el diseño, etc.) y a los usuarios (facilidad de uso y aprendizaje).

De esta forma la ISO 9126 presenta 6 características principales capaces de evaluar la calidad de un software: funcionalidad, fiabilidad, eficiencia, usabilidad, mantenibilidad y portabilidad. De la misma forma que el modelo McCall estas caracteriśticas se dividen en subcaracterísticas cmo se puede observar en la Figura 3.2.

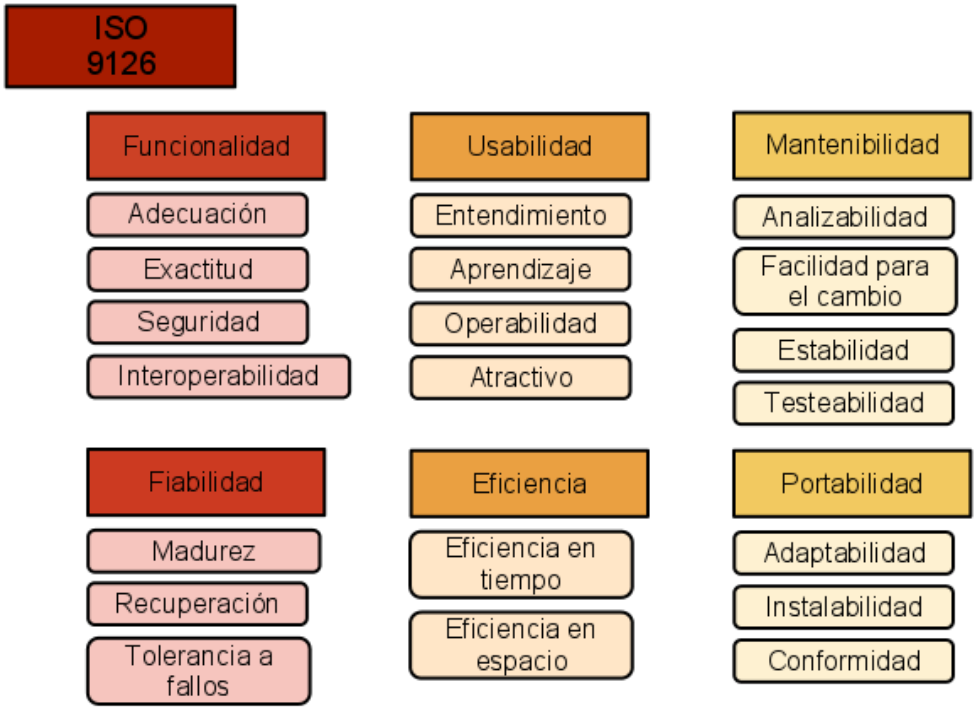

Figura 3.2. ISO 9126

3.2.4. Modelo GQM

El último modelo a comentar es el modelo GQM (Goal-Question-Metric). A diferencia de los modelos vistos anteriormente, este modelo no se basa en características generales de los productos, sino que es aplicable solamente al proyecto en concreto. Toma este enfoque con la idea de que un programa de mesura de calidad puede dar mejores resultados si se diseña con las metas en mente. Simplemente se trata de trabajar realizando tres pasos: crear una lista con las metas del proyecto, a partir de la lista generar unas preguntas que determinen si la meta se ha cumplido y finalmente decidir qué atributos hacen falta medir para responder estas preguntas.

3.3. Garantía de calidad de software sobre MATE

Los objetivos de este proyecto pretenden tomar medidas para garantizar la calidad de software en la aplicación MATE. Para ello se toman como punto de partida los modelos de McCall y la ISO 1926 y se proponen medidas para mejorar los diferentes aspectos de calidad de software que proponen. La elección de dichos modelos se debe a que son capaces de medir de forma cualitativa diferentes aspectos del software en cualquier etapa del ciclo de desarrollo (el uso del model GQM, por ejemplo, requeriría haber sido aplicado desde las etapas más tempranas de desarrollo).

En un sentido general y, principalmente, debido a que MATE ya es un sistema acabado (funcional) centraremos nuestros esfuerzos en los aspectos de calidad restantes. Este proyecto presenta tres lineas de trabajo para garantizar la calidad de software:

- Metodología de desarrollo: proporciona especificaciones sobre aspectos tales como la documentación, la codificación o el control de versiones. Su objetivo es dotar a los desarrolladores con directrices y buenas prácticas de cara a las nuevas lineas de trabajo.

  El conjunto de especificaciones tiene como objetivo favorecer la mantenibilidad a través de la documentación, la trazabilidad a través del control de versiones y a la flexibilidad/capacidad de prueba a través de la especificación sobre desplegamiento y construcción.

- Entorno de desarrollo: provee de herramientas de soporte para la metodología y cubre aspectos de la calidad de software como control de versiones y trazabilidad. Su objetivo es poner al alcance de los desarrolladores las tecnologías necesarias para simplificar el seguimiento de las buenas prácticas propuestas en la metodología y para garantizar otros aspectos de la calidad de software fácilmente.

- Nuevas características: este proyecto propone el desarrollo de cuatro nuevos módulos para mejorar diferentes aspectos:

- Sistema de instalación: provee a MATE de una capacidad de instalación más flexible, busca dependencias, permite escoger características, etc. Esta característica pretende favorecer la instalabilidad y la portabilidad.
- Sistema de configuración: permite a MATE leer ficheros de configuración en diferentes formatos. Esta característica pretende favorecer la usabilidad.
- Sistema de gestión de excepciones de DMLib: permite controlar de forma eficaz las excepciones específicas. Esta característica tiene como objetivo mejorar la fiabilidad, especialmente en lo referente a tolerancia a fallos.
- Sistema de cerrado: permite parar la ejecución del sistema de forma controlada y centralizada. Este sistema tiene como objetivo, por un lado, la usabilidad de cara al usuario y la eficiencia en espacio debido a la limpieza de variables controlada.

4. Definición e implantación de metodología de desarrollo

En este capítulo se presenta la metodología de desarrollo que se propone para el proyecto MATE. Primero se hace una perspectiva general sobre los principales elementos que la conforman (las guías de estilo) y luego se detalla la contribución personal respecto a la confección de éstas. Además se hace un introducción al concepto de prueba unitaria y se comenta su valor dentro de la metodología planteada.

Una vez explicados todos los aspectos de la metodología, se muestra el proceso de implantación de ésta sobre el código actual de MATE. La adaptación a la metodología se muestra en tres fases: adaptación a los cambios de la librería Common, adaptación a guías de estilo (refactorización y documentación) y prueba unitaria.

4.1. Aspectos de la metodología

Una metodología de desarrollo consiste en un conjunto de buenas prácticas con el objetivo de conseguir un producto software de mayor calidad. El conjunto de buenas prácticas que se plantea en este proyecto abarca aquellos aspectos de la calidad más alejados de la funcionalidad y la eficiencia para centrarse en aspectos a veces descuidados como la mantenibilidad y la usabilidad.

4.1.1. Guías de estilo

La metodología que se presenta en este proyecto tiene como objetivo unificar criterios de cara al equipo de desarrollo para poder crear un producto más uniforme.

Para conseguir este objetivo la metodología consta de una serie de guías de estilo, también llamadas especificaciones que abarcan cinco aspectos del software, la codificación, la documentación, el despegamiento, el control de versiones y la construcción.

- Guía de estilo de documentación: esta guía busca sentar unas bases firmes en cuanto a estructura, contenido general y formato para el resto de documentación referida al proyecto MATE.
- Guía de estilo de documentación de código: el objetivo principal de esta guía es definir qué clase de información ha de contener la documentación sobre el código, en especial que elementos (clases, métodos, miembros, etc) han de estar comentados y cómo.
- Guía de estilo de codificación: el propósito de esta guía proporcionar directrices respecto al formato (tipos de fichero, espaciado, indentación, división de lineas, etc) y respecto a prácticas de programación (nombrado de variables, clases, métodos, constantes; visibilidad, etc.).
- Guía de estilo de control de versiones y construcción: este documento proporciona información sobre la dinámica de trabajo respecto al control de versiones y la construcción respecto a las herramientas implantadas para ello (léase sección 5.2.1 Componentes del sistema de desarrollo, para más información)
- Guía de estilo de desplegamiento: esta guía provee instrucciones sobre como el código debe ser preparado para ser distribuido, trata temas como la organización de ficheros fuente, los módulos a generar y los procesos de instalación, actualización y desinstalación.

4.1.2. División de trabajo

Respecto a la confección de las guías, el trabajo se divide de forma que la carga temporal quede balanceada. La tabla 4.1 muestra la relación miembro del equipo de

desarrollo y guías confeccionadas.

Miembro	Guías confeccionadas
Noel De Martin	• Guía de estilo de codificación • Guía de estilo de control de versiones y construcción (construcción)
Rodrigo Echeverría	• Guía de estilo documentación • Guía de estilo de documentación de código
Toni Pimenta	• Guía de estilo de desplegamiento • Guía de estilo de control de versiones y construcción (control de versiones)

Tabla 4.1. División de trabajo respecto a la definición de la metodología

Respecto a la implantación de la metodología, cada miembro del equipo de desarrollo será responsable de aplicar todos los aspectos de la metodología (refactorización de código, documentación, prueba unitaria) a un módulo de MATE. Esta división se efectúa de esta forma debido, principalmente, al hecho que MATE es una aplicación compleja y su documentación y prueba requieren de un cierto grado de "dominio" sobre su funcionamiento, lo hace que sea más productivo que cada miembro se centre en una parte (módulo). La siguiente tabla muestra la relación módulo y miembro asignado.

Miembro	Módulo asignado
Noel De Martin	DMLib Common
Rodrigo Echeverría	Analyzer
Toni Pimenta	AC

Tabla 4.2. División del trabajo respecto a la implantación de la metodología

4.1.3. Guía de estilo de documentación

Como se comentaba anteriormente, esta guía busca sentar unas bases firmes en cuanto a estructura, contenido general y formato para el resto de documentación referida al

proyecto MATE. Esto incluye el resto de guías de estilo, documentos de planificación , manuales de usuario, documentos de casos de prueba, etc. Su objetivo principal es conseguir un alto nivel de homogeneidad en la documentación para hacerla más portable entre los diferentes miembros de la jerarquía de desarrollo (directores de proyecto, analistas, desarrolladores) y, también, los usuarios finales (en el caso de documentación de carácter externo, como las guías de instalación).

El documento está dividido en tres capítulos, el primero está referido a la estructura general que seguirán los documentos, el segundo especifica que clase de información contendrá cada sección y el tercero concreta unas ciertas convenciones de formato para los diferentes elementos que puedan aparecer en los documentos (títulos, apartados, código, tablas, figuras, etc.).

Respecto a la estructura y a al contenido de los documentos en general, esta guía propone una estructura en dos partes, una constante (común a todos los documentos) y el propio cuerpo del documento. La primera parte contiene toda la información referida al propio documento, de forma que el lector sea capaz de encontrar rápidamente la información que busca. Esta parte común consta de la portada (que contiene el título del documento y el control de versiones), el indice y un primer apartado de introducción al documento, que contiene información sobre los posibles interesados, el propósito del documento, su alcance y su estructura.

En cuanto al formato, la guía detalla el uso de tipografías según el contexto (cabeceras, cuerpo de texto, ordenes de terminal, código, etc.); el uso de tablas, figuras y cabeceras; la numeración de secciones y el control de versiones de documentos.

Para más información consultar anexo 1.

4.1.4. Guía de estilo de documentación de código

El objetivo principal de esta guía es proporcionar instrucciones a los desarrolladores acerca de que partes del código se deben documentar, que información se debe dar sobre ellas y como debe estar presentada.

El cuerpo del documento está dividido en tres capítulos, el primero, principios de documentación, que aporta consejos sobre que lineas debe seguir la documentación; el segundo, estructura, que detalla la información que sebe dar a diferentes niveles (paquete, clase y método) y el tercero, *tags*, que explica como utilizar etiquetas para que una herramienta de documentación pueda generarla a partir de los comentarios (léase Doxygen en la sección 5.2.1).

Respecto a las lineas generales, esta guía aconseja, entre otras cosas, el uso del inglés como idioma de la documentación para que ésta sea portable y la independencia de la implementación (es decir que la documentación no detalle cómo están implementadas las funciones o las clases, cosa que se debe hacer en el cuerpo de las mismas).

Respecto a la información que debe aparecer en la documentación, se aconseja que a nivel de módulo/paquete exista información sobre la jerarquía de clases, a nivel de clases que se especifique que datos encapsulan y como trabajar con ellos y a nivel de métodos que se detalle no solo que hacen si no, que parámetros reciben, que retornan, que clase de excepciones pueden lanzar, etc.

En cuanto a las etiquetas, se propone un conjunto interpretable por Doxygen, que permitan discernir con claridad el resumen de contenido, autores, versiones, parámetros, valores de retorno, etc.

Para más información consultar anexo 2

4.2. Aplicación de la metodología

4.2.1. Primera fase: adaptación a Common

La adaptación a las guías de estilo se llevó a cabo en dos fases, en la primera, el objetivo fue adaptar Analyzer a los cambios introducidos por Noel De Marin *[MAR 11]* en la librería Common. Estos cambios fueron hechos para resolver los conflictos (sobretodo relacionados con duplicidad de código) entre Common y DMLib e implicaron que tanto Analyzer como AC, que también dependían de Common, fueran

refactorizados en consecuencia.

La refactorización de Analyzer implicó una serie de cambios menores derivados de la adaptación de Common a la guía de codificación respecto al uso de mayúsculas para nombrar clases. Además, se cambió la llamada a la función que procesa el archivo de configuración de entrada en la clase ctrl, debido al nuevo sistema jerarquizado, basado en objetos, introducido en Common:

```
Common::Config::LoadFromFile //Old
_cfg = ConfigHelper::ReadFromFile(cfgFile);//New
```

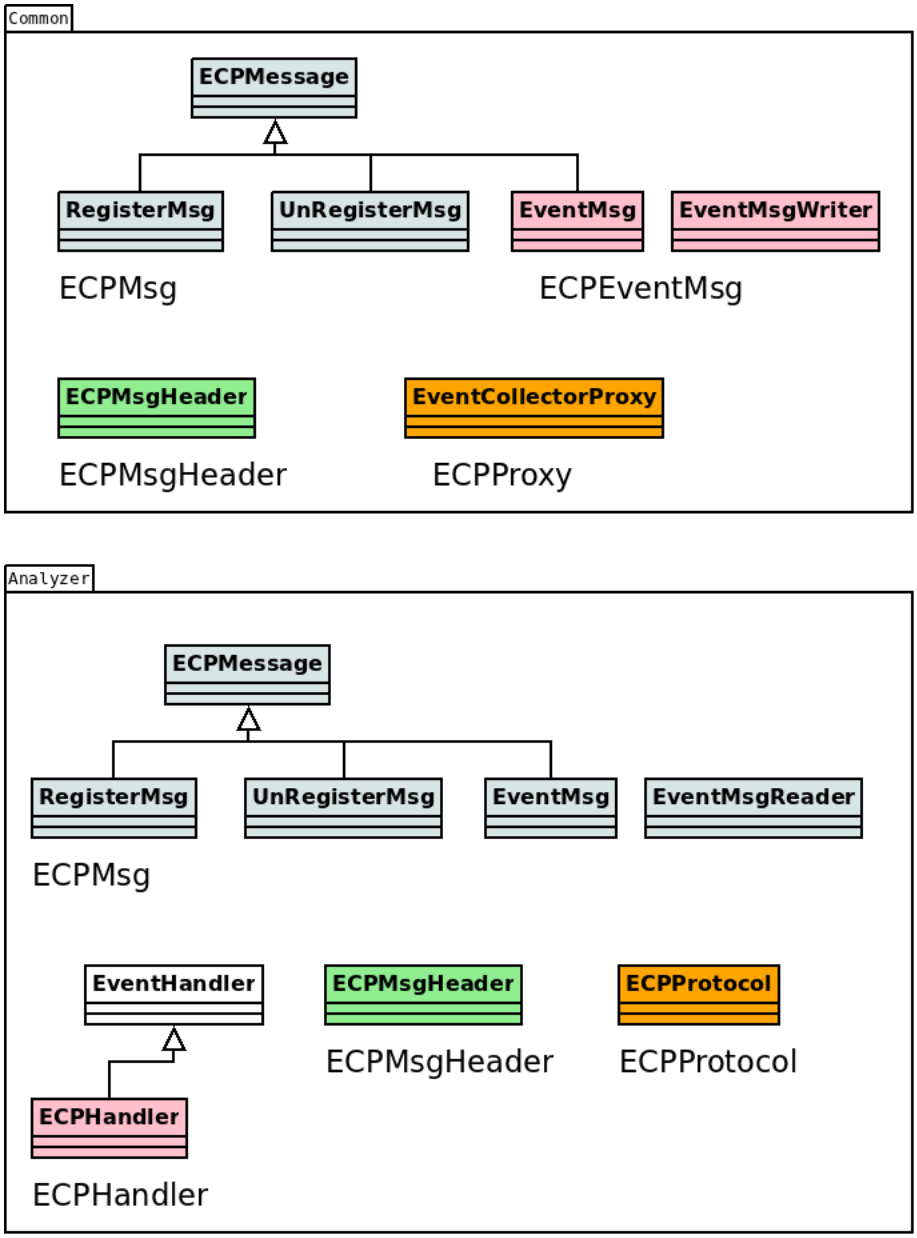

Figura 4.1. Colisiones entre Common y Analyzer

Por otra parte, también respecto a la librería Common, se detectó duplicidad de código respecto a Analyzer en las clases dedicadas al paso de mensajes de eventos, la Figura 4.1 muestra un esquema con los ficheros fuente y las clases implicadas. En la figura, el color común indica que dos clases están en el mismo archivo fuente, cuyo nombre aparece debajo.

La resolución de este conflicto consistió básicamente en mantener en Common

aquellas clases que fueran a ser utilizadas realmente por otros módulos aparte de Analyzer y dejar en este aquellas que estrictamente solo usa él:

- Jerarquía Event Collector Protocol Message (ECPMessage, RegisterMsg, UnRegisterMsg y EventMsg): se quita de Analyzer ya que el módulo que emite los mensajes (DMLib) también necesita saber como encapsularlos y se coloca en Common bajo el nombre ECPMsg.
- EventMsgWriter: se mantiene en Common puesto que es necesario por los módulos emisores, pero se separa del fichero ECPEventMsg a un fichero propio con el nombre de la clase.
- EventMsgReader: se mantiene en Analyzer puesto que éste es el único receptor y, por lo tanto, lector, de mensajes. Se coloca en un fichero aparte llamdo EventMsgReader.
- ECPMsgHeader: se borra de Analyzer y se mantiene en Common, ya que tanto emisor (DMLib) como receptor (Analyzer) deben conocer como son las cabeceras de los mensajes.
- ECPHandler, ECPProtocol: se mantienen como estaban, en Analyzer.
- ECPProxy: se mantiene como estaba, en Common.

4.2.2. Segunda fase: documentación y refactorización

Para adaptar el código a la guía de documentación de código se procedió a añadir documentación a las clases de Analyzer y a sus métodos. Tal y como dicta la guía, esta documentación se hizo sobre los ficheros de cabecera para mantener los ficheros de implementación con el mayor grado de limpieza posible. El siguiente fragmento muestra un ejemplo de documentación aplicada al método *CreateApplication* de la clase *DTLibrary*:

```
/**
 * @brief creates a new application model, a new event
 * collector and associates them both.
```

```
 * @param appPath   path to the target application.
 * @param argc      number of arguments of the target
 *                  application.
 * @param argv      list of arguments of the target
 *                  application.
 *
 * @return     reference to the application model object.
 */
Model::Application & CreateApplication (
                                    char const * appPath,
                                    int argc,
                                    char const ** argv);
```

El proceso de documentación abarca las siguientes clases (y sus métodos), junto a la clase se informa si esta documentada o no y cuantos de sus métodos lo están:

- ACProxy [si] 11/11
- Tunlet [no] 0/4
- AdjustingNWTunlet [no] 0/14
- Event [si] 11/11
- EventRecord [si] 8/8
- EventHandler [si] 1/1
- Events [si] 5/5
- Application [si] 27/27
- Host [si] 2/2
- HostHandler [si] 2/2
- Task [si] 20/21
- TaskHandler [si] 2/2
- Tasks [si] 9/9
- CommandLine [si] 11/11
- Controller [si] 2/2
- DTLibrary [si] 4/4
- DTLibraryFactory [si] 2/2
- ECPHandler [si] 6/6
- ECPProtocol [si] 2/2
- EventListener [si] 2/2
- ECPAcceptor [no] 5/5
- EventCollector [si] 9/9
- EventMsgReader [si] 10/10
- WorkerData [si] 0/17
- BatchData [si] 0/18
- IterData [si] 0/13
- FactoringTunlet [si] 1/18
- Service [si] 4/4
- ShutDownManager [si] 7/7
- TunletsContainer [si] 0/0

El resultado de esto es que la cobertura de clases es del 90% (27/30) y la de métodos del 66,53% (163/245), aunque cabe destacar que la mayoría de los métodos no comentados pertencen a la implementación de los tunlets, y para comentarlos se necesitaría más formación en computación de altas prestaciones. El resultado de la documentación se puede observar en el anexo 4.

Para adaptar el código a la guía de codificación se procedió a cambiar los siguientes

aspectos en las miscmas clases que la documentación:

- Tamaño máximo de línea de entre 80 y 100 caracteres.
- Nombres de variable más explícitos.
- Inglés como idioma para nombres de variables, estructuras, constantes, etc.
- Uso de constantes en los lugares necesarios y capitalización de sus nombres.
- Eliminación de saltos de linea antes de llaves de apertura en clases, métodos y sentencias de control.
- Espaciado entre componentes de expresiones.
- Eliminación del espaciado entre el nombre y los parámetros en las llamadas a función.
- Sustracción de las llaves innecesarias (sentencias de control con una única sentencia).
- Añadido de comentarios junto a llaves de cerrado en casos de ambigüedad.
- Utilización del operador ternario `(condición) ? expressión1 : expressión2;` en los casos oportunos.

5. Implantación del entorno de desarrollo

En este capítulo se muestra como se llevó a cabo la implantación de un sistema de desarrollo para el proyecto MATE. Primero se detallan las características buscadas para el sistema y se presenta una visión de conjunto de las herramientas escogidas para satisfacerlas y se especifica la división de trabajo entre los miembros del grupo.

Luego, se detallan las características de la herramienta asignada y se muestra el proceso de implantación: instalación, confección del manual de instalación y configuración, integración y automatización del proceso.

5.1. Especificación del entorno de desarrollo

El entorno de desarrollo tiene dos objetivos principales, por una parte, proveer tecnologías de soporte para el seguimiento de las practicas propuestas en la metodología presentada en el capítulo anterior y, por la otra, proporcionar herramientas para la gestión del proyecto y para la comunicación entre los miembros del equipo de desarrollo.

5.1.1. Características funcionales

- Gestión de proyecto: el sistema ha de proveer mecanismos para la gestión de las diferentes lineas de proyecto: planificación, gestión de miembros y equipos, sistemas de colaboración, trazado de características, etc.
- Comunicación: el entorno ha de proporcionar mecanismos de comunicación entre los miembros involucrados en el proyecto. El objetivo es que esta comunicación sea lo más efectiva posible respecto a los cambios realizados sobre código, diseño, documentación, etc para mejorar la productividad y la efectividad del equipo.
- Control de versiones: el sistema ha de proveer mecanismos para mantener un

registro sobre las modificaciones introducidas en las versiones del código fuente, así como permitir la coexistencia (*branching*) de diferentes versiones y proporcionar mecanismos para combinar los cambios realizados en ellas (*merging*).

- Automatización de la construcción: para mejorar el rendimiento de trabajo se han de proporcionar herramientas capaces de recompilar, reenlazar y ejecutar MATE en diferentes entornos objetivo de forma automática y centralizada.
- Documentación: el entorno tiene que proporcionar un mecanismo para generar documentación de forma automática a partir de los comentarios hechos sobre código (consultar sección 4.1.4 para más información).

5.1.2. Características no funcionales

- Transportabilidad: respecto al software que cumple los diferentes requerimientos funcionales, el sistema ha de ser fácilmente reproducible en otro entorno hardware (para facilitar tareas de mantenimiento o de actualización del propio entorno).
- Instalación automática: tiene que existir un sistema capaz de instalar los componentes principales del entorno e integrarlos de forma automática.
- Información centralizada: la mayor parte del entorno de desarrollo ha de ser centralizada por dos motivos, primero para mantener la coherencia de la información (documentación, código,etc) y segundo porque la compilación y prueba de MATE se realiza sobre un cluster de computadores y, por lo tanto, un único sistema ha de tener acceso a él.

5.2. Especificación del entorno de desarrollo

El entorno de desarrollo seguirá una arquitectura cliente-servidor. El servidor tendrá acceso por medio de una red de área local a un cluster de computadores (que actuará como infraestructura de pruebas) y a una SAN (Storage Area Network), donde almacenará archivos como el código fuente de las diferentes versiones. Además, funcionará como punto de acceso a esta infraestructura a las máquinas cliente, a las que presentará una interfaz de trabajo donde gestionar el proyecto, desarrollar, probar, etc.

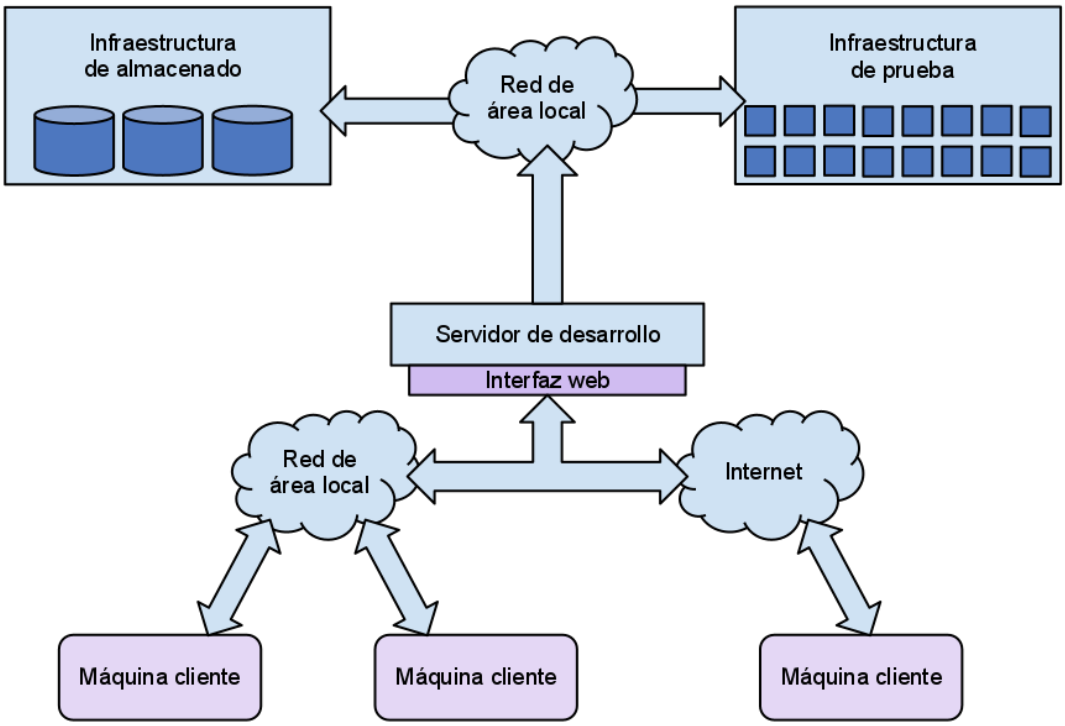

Figura 5.1. Infraestructura de desarrollo

La Figura 5.1 presenta un ejemplo de entorno de desarrollo como el descrito anteriormente, al que se conectan mediante una conexión de área local dos máquinas cliente y de forma remota, a través de Internet, otra.

5.2.1. Componentes del sistema de desarrollo

Para satisfacer los requisitos planteados en la sección anterior se seleccionan los

siguientes componentes:

- Sistema Operativo Base: Ubuntu 10.04 LTS (*Long Term Support*) [13]
- Servidor web: Apache HTTP Server [3]
- Herramienta de colaboración: Redmine 1.1 [4]
- Herramienta de control de versiones: Subversion (SVN) [6]
- Herramienta de construccion: Buildbot [7]
- Herramienta de documentación: Doxygen [8]
- Entorno cliente: Eclipse Helios (*propuesta*) [9]

Ubuntu 10.04 LTS actuará como sistema operativo base para el resto de software corriendo en el servidor. Los motivos para seleccionar este sistema son los siguientes:

- Capacidad de integración: al tratarse de un sistema UNIX (GNU/Linux) posee con diferentes implementaciones del estándar MPI, la API Dyninst, Servidores web y sistemas de creación de máquinas virtuales.
- Reducción de costes: Ubuntu es totalmente gratuito para su uso y distribución.
- *Long Term Support*: la versión 10.04 de Ubuntu es una versión con soporte a largo termino, lo que significa que recibirá actualizaciones por parte de Canonical durante 3 años.
- Familiaridad: Ubuntu es uno de los sistemas operativos GNU/Linux con más difusión y posee una gran comunidad de usuarios que lo respaldan y que plantean soluciones en linea para la mayoría de problemas que puedan surgir.

Apache HTTP Server será el que maneje las peticiones de las máquinas cliente, proporcionará acceso, por una parte, a la interfaz web de Redmine y, por la otra, a la interfaz web de Subversion (Repositiorio).

Redmine será el encargado de la gestión del proyecto, servirá, por una parte, como herramienta de planificación de tareas (integrándose con el repositorio de código del

lado del servidor y con el entorno de desarrollo del lado del cliente) y, por otra, como herramienta de comunicación entre miembros a través del uso de foros/wikis.

Subversion es la herramienta encargada de gestionar el repositorio de código, automatizará el proceso de control de versiones, gestionará el acceso concurrente y automatizara el proceso de *branching-merging* entre versiones. Además estará integrado con Apache Server para que el repositorio pueda ser consultado mediante un navegador web.

Buildbot es la herramienta encargada de la automatización de la construcción y el test, sus objetivos principales son dos, construir automáticamente cada vez que se realicen cambios para detectar fallos lo más pronto posible y construir en diferentes entornos (máquinas virtuales) de forma remota.

Doxygen es la aplicación que se utilizará para generar la documentación de MATE de forma automática a partir de los comentarios de documentación que hay sobre el código.

Eclipse es un entorno de desarrollo integrado que proponemos como entorno cliente. Uno de sus mayores potenciales es la capacidad de ampliación mediante *plug-in's*: CDT para el desarrollo en C/C++, ECUT/CUTE para la prueba unitaria, eclox para la integración con Doxygen, mylyn para la integración con Redmine, etc. Por otra parte su capacidad de uso en el desarrollo de aplicaciones paralelas (como MATE) tendrá que ser evaluada más adelante por desarrolladores más experimentados y, queda excluida del alcance de este proyecto.

5.2.2. Integración de componentes

Los principales objetivos de un entorno integrado son mejorar la productividad y la eficacia del equipo de desarrollo es por esto que el grado de cohesión entre las herramientas ha de ser lo más alto posible, esto es, en nuestro caso, que exista un enlace entre un elemento planificado con la versión de su código fuente correspondiente y que esta pueda ser compilada, enlazada y ejecutada en diferentes entornos de forma automática y, una vez probada, si existiese un problema o bug este

sea reportado e incluido en la planificación.

La Figura 5.2 es un diagrama de integración de los diferentes componentes propuestos para nuestro entorno de desarrollo. La integración se llevará a cabo de la siguiente manera: tanto Redmine como SVN correrán en servidores virtuales de Apache y serán visibles por parte del entorno cliente (de esta forma el usuario podrá consultar la planificación, comunicarse, etc. a través de Redmine y también podrá ver el contenido del repositorio a través de SVN). Por otra parte Redmine y SVN estarán comunicados de forma que se puedas asociar elementos de la planificación con versiones de software en el repositorio. Además SVN y Buildbot estarán conectados de forma que se pueda efectuar la construcción de versiones de código concretas de forma automática. Con respecto al entorno cliente, además de poder conectarse al servidor mediante un navegador, podrá conectar Eclipse a Redmine mediante un *plug-in* (MyLyn) y podrá mirar las tareas que tenga planificadas y trabajar en local con el código fuente.

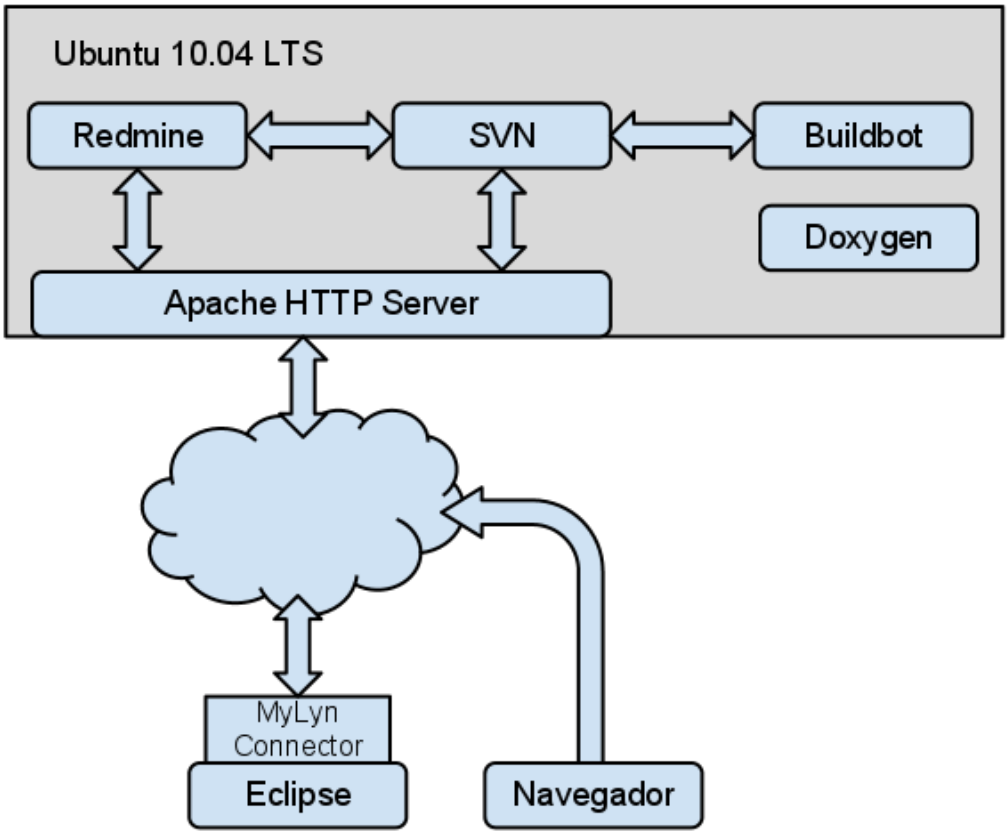

Figura 5.2. Entorno de desarrollo

5.2.3. Distribución del trabajo

La implantación de cada herramienta* consta de cuatro fases: instalación, confección de una guía de instalación y configuración, integración y automatización del proceso de instalación mediante un script. La división de la carga de trabajo entre los miembros del equipo se muestra en la Tabla 5.1.

** Exceptuando Eclipse, que solo es una propuesta de entorno cliente (léase sección 5.2.2 para más información).*

Miembro	Tareas asignadas
Rodrigo Echeverría	• Instalación de Redmine • Guía de instalación y configuración de Redmine • Integración de Redmine con Apache • Integración de Redmine con Eclipse • Script de instalación de Redmine
Toni Pimenta	• Instalación de Apache • Instalación de SVN • Instalación de Doxygen • Integración de Apache y SVN • Guía de instalación y configuración de Apache y SVN • Integración de SVN y Redmine • Script de instalación de SVN • Script de instalación de Doxygen
Noel De Martin	• Instalación de Buildbot • Guía de instalación y configuración de Buildbot • Integración de SVN y Buildbot • Script de instalación de Buildbot

5.1. División del trabajo respecto a la implantación del sistema de desarrollo

5.3. Redmine

Redmine es una herramienta dedicada a la gestión y planificación de proyectos con interfaz web 2.0 basada en el framework Ruby on Rails. Uno de sus puntos a favor más significativos es su usabilidad, ya que basa su funcionalidad en una interfaz web clara y sencilla basada en pestañas, fácil de aprender, configurar y personalizar. La Figura 5.3 muestra un ejemplo de la interfaz de Redmine mostrando un diagrama de Gantt.

Destaca sobre su principal alternativa, Trac [5], en que, mientras que Redmine está preparado por defecto para ser una solución completa, Trac requiere de un proceso de configuración de múltiples *plug-in's* para ofrecerla. Por otra parte, Redmine ha demostrado ser un proyecto con un desarrollo más activo que Trac, siendo actualizado aproximadamente cada dos meses frente a éste, cuya última versión estable es de junio de 2010.

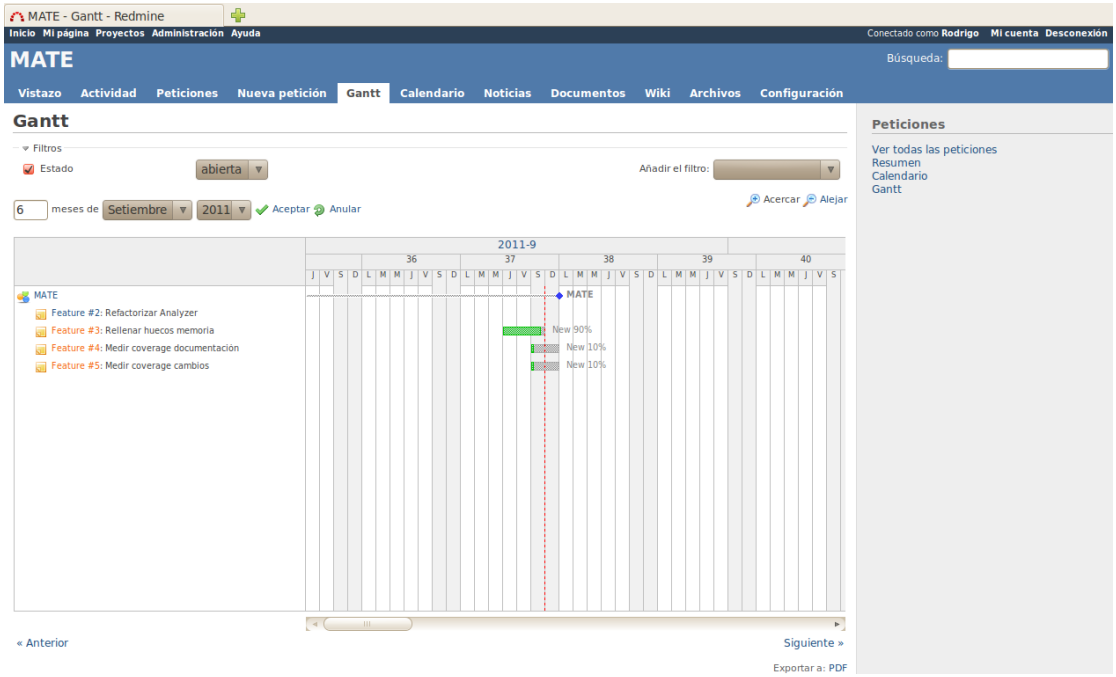

Figura 5.3. Captura de la interfaz de Redmine

5.3.1 Características

- **Soporte multiproyecto**: Redmine permite gestionar diferentes proyectos desde una misma interfaz, cada uno de los cuales puede poseer su propia configuración, es decir, que módulos de Redmine va a utilizar (calendario, diagramas de Gantt, foros, *wikis*, repositorios, etc.). Además aporta opciones de privacidad para cada uno de ellos, puede ser visible al público o puede limitar la visibilidad de características basándose en roles (que, además se pueden gestionar de forma diferente para cada proyecto.

- **Gestión de usuarios basada en roles**: Redmine controla el acceso a los diferentes módulos de un proyecto basándose en roles, no en usuarios individuales. El rol es el papel que juega el usuario dentro del proyecto, Redmine soporta que un usuario pueda poseer más de un rol (o, lo que es lo mismo, que pertenezca a más de un grupo). El sistema de roles es completamente configurable de cara al proyecto, permitiendo no solo asignar

roles a los diferentes usuarios, sino crear los roles que más se adecuen al proyecto y asignarle a estos sus niveles de privacidad correspondientes.

- **Trazado de unidades de trabajo**: La unidad de trabajo de Redmine se denomina petición (issue, en inglés) y la planificación de proyectos se basa en ella. Las peticiones pueden clasificarse en tres tipos básicos, tareas, errores y soporte, que pueden ser ampliados en la configuración. Las peticiones contienen toda la información necesaria para su planificación: miembros asignados, fecha de inicio, fecha de finalización, nivel de prioridad, porcentaje completado, etc. y se pueden enlazar con la subida de un fichero.
- **Diagramas de Gantt y calendarios**: Redmine incluye un calendario para visualizar todas las peticiones mostrando su inicio y su final, además posee una vista de la misma información en un diagrama de Gantt que indica además el porcentaje completado. Para estos dos módulos Redmine permite el filtrado de las peticiones que se visualizan en función de sus características (fechas de inicio y fin, nivel de prioridad, porcentaje completado, usuario asignado, etc.).
- **Gestión de noticias, documentos y archivos**: El entorno permite crear noticias como medio de difusión para los usuarios. Además permite y gestiona la subida de documentos y archivos.
- **Notificaciones por e-mail**: Configurando el servidor SMTP (*Simple Mail Transfer Protocol*), Redmine permite configurar eventos para que, al suceder, envíen una notificación por correo a los usuarios. Además los usuarios pueden configurar que clase de notificaciones recibir, por ejemplo, solo aquellas relacionadas con peticiones en las que estén involucrados.
- **Soporte para la creación de *wikis* y foros**: Redmine posee dos módulos destinados a la intercomunicación de los miembros de un proyecto: *wiki* y foros. Las *wiki* (del hawaiano, rápido) son páginas web escritas de forma ágil a través del navegador y que pueden ser modificadas por los diferentes miembros del equipo de desarrollo, convirtiéndose así en un medio idóneo para compartir información, documentación, manuales, etc. Los foros por su

parte permiten que diferentes miembros colaboren en las mismas peticiones y les permiten compartir información y dudas sobre las mismas.

- **Integración con sistemas de gestión de la configuración de software (SCM)**: Como se comentaba en el apartado de integración, Redmine se puede integrar con sistemas de configuración de software como Subversion, de esta forma es posible asociar peticiones a las fuentes de código del repositorio que las cumplen (y, sobretodo, en que versión lo hacen).
- **Soporte de idiomas**: Redmine cuenta con un extenso soporte de idiomas, entre los que figuran el inglés (por defecto), el castellano y el catalán. Además de permitir la asignación de un idioma por defecto para la aplicación y para el proyecto, permite que cada usuario pueda verla en el idioma que prefiera.
- **Soporte para el desarrollo e integración de plugins**: Además de todas las características mencionadas, Redmine permite la ampliación de su funcionalidad con *plug-in's* desarrollados por la comunidad. Entre las decenas de plug-in's destacan aquellos para generar diagramas (charts), creación de salas de chat y gestión de proyectos Ágiles.

5.3.2. Garantía de calidad de software con Redmine

Además de la gestión íntegra del proyecto, Redmine aporta soluciones para dos de los mayores problemas relacionados con la garantía de calidad del software: el problema de la comunicación y el problema del trazado de características.

El primer problema, el problema de la comunicación está relacionado con el control de versiones y el acceso concurrente al mismo código. Si bien los sistemas de gestión de la configuración de software permiten que las transacciones se puedan acompañar de un comentario que indique que cambios se han realizado sobre el software esto simplemente mitiga algunos problemas pero no es una solución completa.

La comunicación efectiva entre miembros permite no solo controlar los cambios hechos en el código, sino también hacer división de tareas y colaboración en la

consecución de las mismas. De esta forma se consigue, por un lado, un grado de productividad óptimo, ya que cada miembro trabaja sobre características o problemas diferentes al mismo tiempo y, por el otro, un grado de efectividad más alto puesto que queda controlado de antemano el acceso concurrente al código (se evitan situaciones donde los cambios de los desarrolladores son inconsistentes o incompatibles entre si).

Para mejorar aspectos de la comunicación Redmine aporta multitud de herramientas/módulos: gestión de noticias, foros, wiki, aparte del calendario accesible para todos los miembros y en el que se puede detallar para cada tarea en que aspecto están implicados los miembros encargados de llevarla a cabo.

El segundo problema, el problema del trazado de características involucra a todas las etapas del proceso de desarrollo software, y se refiere al grado en que cada etapa satisface a la anterior y, en consecuencia, en que grado el producto final se corresponde con las especificaciones iniciales. El problema del trazado ha de resolverse en las diferentes etapas del desarrollo; el análisis de requerimientos se tiene que corresponder con los requerimientos, el diseño se tiene que corresponder con el análisis de requerimientos, el código se tiene que corresponder con el diseño y los casos de prueba con el código.

Para mitigar este problema existen dos prácticas básicas, la primera es el uso de modelos de desarrollo iterativos que tengan comunicación con el cliente para poder detectar anomalías de forma temprana y la segunda es apoyar el proceso de desarrollo en herramientas que permitan el trazado de características.

Redmine, en este caso, aporta características capaces de apoyar a ambas prácticas, por una parte, todas las herramientas destinadas a la comunicación descritas anteriormente, que permiten, mediante la asignación de un rol al cliente, la capacidad de entrar directamente en el proceso de desarrollo y, por la otra permiten el seguimiento de características a través de todo el ciclo de desarrollo gracias a la capacidad de integración de las peticiones tanto con documentos como con versiones de código fuente.

5.3.3. Instalación y configuración de Redmine

La instalación de Redmine es un proceso tedioso puesto que involucra el uso de dependencias en versiones especificas, la creación de bases de datos y el manejo de variables de entorno de Ruby.

En este caso se instalara Redmine en su versión 1.1, la más reciente hasta la fecha. El primer paso consiste en instalar sus dependencias directas relacionadas con Ruby en las siguientes versiones:

- Ruby 1.8.7: lenguaje de programación en que está escrito Redmine.
- RubyGems 1.3.7: gestor de paquetes (gemas) de Ruby.
- RubyOnRails 2.3.5 (gema): framework para aplicaciones web escrito en Ruby.
- Rack 1.0.1 (gema): interfaz con servidores para aplicaciones web.

Para instalar Ruby y RubyGems existen dos opciones: instalarlos mediante el gestor de paquetes de Ubuntu (aptitude) o bajar el código fuente, compilarlo y enlazar el ejecutable en un directorio por defecto (/usr/local/bin, por ejemplo). En una primera versión de la instalación (y de la guía de instalación) se hizo mediante e gestor debido a un bug en el código fuente pero, pasado el tiempo, una vez corregido el bug, se optó por instalarlo manualmente ya que es la mejor forma de controlar que se instala la versión de código deseada. La instalación manual, requiere ademas que se satisfagan las dependencias de Ruby antes, en un sistem Ubuntu acabado de instalar estas son: libssl-dev libopenssl-ruby1.8 y libc6-dev, que se pueden instalar con el gestor de paquetes.

Una vez instalado RubyGems, se puede utilizar para instalar las gemas de RubyOnRails y Rack en la versión adecuada a través de él.

Una vez instaladas las dependencias con Ruby, cabe instalar las dependencias con mySQL y preparar una base de datos para Redmine. Suponiendo que mySQL ya se encuentra instalado, las dependencias de Redmine de cara a éste son el paquete libmysql-ruby (instalable a través de cualquier gestor de paquetes) y la gema mysql

(instalable mediante rubyGems como anteriormente). Una vez instaladas las dependencias con mysql se debe crear una base de datos, se debe abrir una consola mySQL, crear una base de datos (llamada redmine por ejemplo) y un usuario con privilegios para ésta.

Una vez instaladas las dependencias, se puede descargar desde su repositorio la versión 1.1 de Redmine utilizando SVN. Una vez descargada la aplicación, se debe crear un archivo de configuración adecuado para la base de datos en la carpeta config (dentro de esta hay un patrón de ejemplo llamado database.yml.example), este archivo de configuración se ha de rellenar con los datos de la base de datos mySQL y el usario que creamos anteriormente. Una vez configurada la base de datos solo resta llenarla con una configuración por defecto mediante los siguientes comandos:

```
$ rake generate_session_store
$ RAILS_ENV=production rake db:migrate
$ RAILS_ENV=production rake redmine:load_default_data
```

Por último basta con crear las carpetas para logs, temporales, plugins y archivos que utilizará Redmine y otorgarle privilegios a éste para que pueda escribir en ellas.

5.3.4. Integración de Redmine

Integración con Apache

Para que Redmine sea totalmente funcional, esto es, para que sea visible por maquinas cliente y no solo en local, se debe integrar con Apache. Una opción para hacer esto es instalar el módulo phusion passenger [10] (también conocido como mod_rails o mod_rack) en Apache, passenger destaca por estar mantenido actualmente (a diferencia de su competencia directa mod_ruby) y por estar probado mediante pruebas de estrés exhaustivas. Para instalarlo basta con instalar su gema (passenger) y usar ésta para instalar el módulo en Apache (passenger-install-apache2-module), es posible que queden dependencias por satisfacer, pero el instalador nos advertirá de ellas y nos sugerirá como instalarlas.

Para acabar la integración basta con crear los archivos de configuración en Apache (entre ellos el servidor virtual que alojará a Redmine). Los archivos de configuración son 3, por motivos de brevedad solo se enunciará el tipo de información que contendrán, pero están disponibles junto al script de instalación en los anexos:

- passenger: le indica a Apache la ruta a la librería dinámica de passenger y pone algunos variables como la cantidad de peticiones máximas a sus valores por defecto.
- redmine.conf: le ndica a apache la ruta al directorio Public de Redmine (punto de partida para la aplicación desde el cliente)
- virtualhost_redmine: crea un servidor virtual usando el puerto 80 y lo configura importando redmine.conf.

Integración con Eclipse

Para integrar Redmine con Eclipse se puede utilizar un plug-in para el manejo de tareas que éste último lleva por defecto, Mylyn, debidamente configurado. Existen básicamente dos alternativas para integrar Mylyn con Redmine, utilizar un conector especializado o un conector web genérico debidamente configurado, se opta por la primera, puesto que es más sencilla y está mejor soportada.

Redmine-Mylyn-Connector es un plug-in de Eclipse que hace de conector con Redmine y puede ser instalado como la mayoría de plug-in's de Eclipse, en el menú "`help → install new software...`" y agregando la dirección del repositorio de desarrollo como fuente de software:

http://redmin-mylyncon.sourceforge.net/update-site/N/

5.3.5. Configuración de Redmine

Por razones de brevedad, en esta sección solo se listaran algunas de las características que redmine permite configurar y como, para más información se puede recurrir a la Guía de instalación y configuración de Redmine (Anexo 3).

- Proyectos
  - Nombre: nombre para el proyecto, no hace falta que sea único.
  - Proyecto padre: permite crear jerarquías de proyectos.
  - Descripción: breve descripción sobre el proyecto.
  - Identificador: identificador único para el proyecto.
  - Módulos: características de Redmine que utiliza (Wiki, Gantt, Prticiones, Noticias, Repositorio, etc.).
  - Tipos de peticiones: clases de peticiones que se pueden hacer (léase sección 5.3.1, apartado trazado de unidades de trabajo para más información).
- Usuarios
  - Identificador: nombre que usa el usuario para acceder a la aplicación.
  - Contraseña: palabra secreta para acceder a la aplicación.
  - Nombre y apellido
  - Correo electrónico
  - Idioma: idioma en que se le presenta la interfaz.
- Perfiles y permisos
  - Proyecto: permisos sobre la creación, modificación, gestión de miembros, administración de módulos y creación de subproyectos.
  - Foros: permisos sobre administración; envío, modificación y borrado de mensajes, tanto própios como agenos.
  - Calendario: visible si/no.
  - Documentos y ficheros: permisos de solo lectura o administración
  - Gantt: visible si/no

  - Peticiones: permisos relativos a administración, visualización, creación, borrado, entre otros.
  - Noticias: comentar si/no, administrar si/no
  - Repositorio: permisos de lectura, escritura y administración.
  - Wiki: permisos de visualización, modificación, administración y borrado, entre otros.
- Configuración (de la aplicación)
  - Título de la aplicación
  - Texto de bienvenida
  - Tamaño máximo de ficheros
  - Nombre y ruta del servidor

5.3.6. Guía de instalación y configuración

De cara a los futuros desarrolladores de MATE, particularmente aquellos que tengan que implantar el sistema final, reinstalarlo o actualizarlo, se ha creado una guía de instalación y configuración que detalla los posos necesarios para instalar, integrar y configurar Redmine.

Esta guía no solo aporta las instrucciones de terminal necesarias, además detalla cada una de ellas cualitativamente, para que el lector sepa en todo momento qué está haciendo exactamente y, en caso de tener algún problema, poder buscar información de forma más eficaz. A parte de lo ya mencionado, para el caso de la instalación, describe el contenido exacto de los archivos de configuración necesarios.

Por otra parte, el manual de configuración, destinado a administradores de la suite, detalla como llevar a cabo las acciones más frecuentes: creación de usuarios, proyectos, roles, asignación de permisos, etc.

Para más información consultar anexo 3.

5.3.7. Automatización del proceso de instalación

El propósito de automatizar el proceso de instalación es el de poder reproducir el sistema con facilidad en caso de migraciones, actualizaciones, etc. Para cumplir este objetivo se propone la creación de un shell script que se encargue del proceso. Este script consistirá básicamente en tres fases: resolución de dependencias, instalación e integración.

Resolución de dependencias

Para resolver el problema de las dependencias existen tres aproximaciones de cara al usuario:

- Instalar las dependencias automáticamente: esta es la solución más sencilla conocidas las dependencias, pero tiene la contrapartida que no permite al usuario escoger que versión de la dependencia quiere o donde instalarla, en un entorno como este donde el grado de cohesión es alto este factor puede llegar a ser especialmente pernicioso.
- Buscar si están instaladas y preguntar al usuario: esta es, quizás, la alternativa más equilibrada, pero pruebas de uso han demostrado que aumenta demasiado la presencia del usuario en el proceso de instalación, cuando el objetivo es, precisamente, minimizarla.
- Buscar si están instaladas, informar de las que falten y cerrar: el objetivo de esta alternativa es el de permitir al usuario instalar las dependencias como crea conveniente, una forma de que sea más completa y eficaz es, en lugar de simplemente informar de las dependencias que faltan, mostrar, también, sugerencias de como instalarlas.

Para el script se ha utilizado la última aproximación debido a la necesidad de poder escoger las dependencias, pero se ha hecho una excepción respecto a las dependencias directas relacionadas con Ruby (que tienen que ser versiones específicas), para estas se ha habilitado una opción en la llamada del script para que el usuario pueda delegar en el script la responsabilidad de instalar Ruby y las gemas en sus versiones correctas,

de esta forma, si el usuario llama al script

```
installRedmine -r
```

éste se encargara de instalar las dependencias relacionadas con Ruby. Además está alternativa no requiere que el usuario esté pendiente del script, puesto que toma la decisión en la llamada.

Instalación

La instalación consiste en los mismos pasos expuestos en la sección 5.3.3, pero, para reducir la parecencia del usuario, se proveen, junto con el instalador, los archivos de configuración necesarios ya preparados. Además se resuelve la etapa de consola MySQL colocando las consultas en variables y pasándoselas a MySQL al iniciar (evitando que el usuario tenga que intervenir manualmente, excepto para poner la contraseña):

```
#Create db for redmine
q1='create database redmine character set utf8;'
q2="grant all privileges on redmine.* to
'redmine'@'localhost' identified by 'redmine';"
q3='quit' #This query will give back the control
          #to this script
mysql -u root -p << eof
$q1
$q2
$q3
eof
```

Integración con Apache

La etapa de integración consiste en los mismos pasos descritos en 5.3.4 y al igual que en la instalación se proporcionan junto al script los archivos de configuración preparados. Además para que el usuario pueda integrar Redmine de otra forma, se le proporciona una opción para la llamada del script -wa (without Apache).

6. Desarrollo de nuevas características

En este capítulo se muestra el desarrollo de dos nuevas características para MATE, el instalador y el módulo de cerrado. La información está presentada siguiendo una estructura similar a la de un modelo de desarrollo clásico (en cascada): análisis de requerimientos, diseño, codificación y prueba.

Puesto que ambos módulos fueron desarrollados por dos miembros del equipo de desarrollo se detalla la división del trabajo en el momento que las lineas de trabajo se bifurcaron.

6.1. Instalador

Actualmente el proceso de compilación, enlazado e instalación de MATE se basa en un archivo de configuración estático (make.conf) y de una serie de makefiles que lo importan.

El archivo de configuración contiene una serie de constantes con toda la información referente a opciones de compilación y directorios de librerías y cabeceras de dependencias.

Los makefiles contienen 4 objetivos, compilación de los ficheros objeto a partir de cada fichero fuente, enlazado para crear el binario/librería objetivo, instalación (copiado) de los ficheros generados y las plantillas de los archivos de configuración (de MATE) y limpieza; además de la opción de ejecutarlos todos consecutivamente (opción por defecto).

La estructura de makefiles es jerárquica (cómo es habitual en sistemas UNIX) y consta de una serie de makefiles, uno para cada módulo y otro en el nivel superior para coordinarlos. El usuario invoca el que se encuentra en el nivel superior con el objetivo deseado y este se encarga de propagar la orden entre los de nivel inferior.

6.1.1. Especificación de requerimientos

Respecto a la configuración, el objetivo es mejorar la facilidad de uso para el usuario final, para ello se dinamizará el proceso de configuración.

Para conseguir esta dinamización se busca, por una parte, automatizar la búsqueda de dependencias y, por otra, ofrecer una pequeña interfaz de usuario que le permita definir las opciones que prefiera manualmente. La lista de requerimientos de configuración se detalla a continuación:

- Soporte a usuario (opción -help)
- Opciones de selección
  - Compilador de C (opción -cc).
  - Compilador de C++ (opción -c++).
  - Directorio de cabeceras de MPI (opción -mpiinc).
  - Librerías de MPI (opción -mpilib).
  - Librería especifica de MPI (opción -mpilibrary).
  - Directorio de cabeceras de Dyninst (opción -dyninstinc).
  - Librerías de Dyninst (opción -dyninstlib).
  - Directorio de librerias de Dwarf (opción -dwarf).
  - Directorio de librerias de Binutils (opción -with-binutils).
  - Makefile objetivo (opción -with-make).
  - Binario de Doxygen (opción -with-doxygen).
  - Arquitectura del sistema (opción -arch).
- Control de dependencias.
- Búsqueda y configuración de dependencias en función de la arquitectura.

6.1.2. División del trabajo

En el desarrollo de esta característica trabajan dos miembros del equipo de desarrollo, Rodrigo Echeverría, autor del presente documento, y Toni Pimenta *[PIM 11]*. El primero desarrolla el script de configuración y el segundo desarrolla el conjunto de MakeFiles y casos de prueba. Para que la división de trabajo permita trabajar en paralelo se acuerda en tiempo de diseño el formato para la salida del script de configuración de forma que se pueda trabajar en los MakeFiles de forma independiente.

Los casos de prueba y sus resultados se presentan en este capítulo, a pesar de no haber sido desarrollados por el autor del presente documento, con motivos académicos, puesto que también demuestran la funcionalidad de las características desarrolladas.

6.1.3. Diseño *script* de configuración

En el proceso de instalación de programas a partir de código fuente, el script *configure* se encarga de buscar las librerías de las que depende el programa y de dar opciones para forzar valores, escoger parámetros o utilizar ciertos módulos. Una vez satisfechas las dependencias y decididas las opciones de construcción, el script pasa esta información a los archivos MakeFile a través de un archivo de configuración o definiendo variables de entorno.

Existen dos alternativas para la producción de configure scripts, manual y automática basada en GNU autotools. GNU autotools consiste en un conjunto de herramientas para generar archivos de instalación (configure y makefile) de forma sencilla y, sobretodo, respetando todos los estándares. *[LUQ 00]*

El uso de autotools tiene, principalmente, dos ventajas, la automatización del proceso y el seguimiento de estándares, lo que tiene una implicación directa sobre la portabilidad. En contrapartida, el proceso es totalmente cerrado, el script se ha de generar mediante una plantilla con opciones limitadas y una vez generado el código es muy complejo e ininteligible y, por lo tanto, muy poco extensible.

Para el script de configuración de MATE se opta por una generación manual del script por dos razones, para ajustarlo con mayor facilidad a los requisitos planteados y para acercar los resultados al modelo de usuario existente, es decir, generar un archivo de configuración similar al que se utilizaba antes.

Para el diseño general del script se toma como referencia el utilizado por la aplicación TAU (*Tuning and Analysis Utilities*) [11], puesto que es una aplicación que comparte dependencias y opciones con MATE. De esta forma, el proceso de configuración se divide en cuatro pasos: colocar las opciones que el usuario ha seleccionado manualmente, determinar la arquitectura del sistema, definir las opciones que no ha especificado el usuario a partir de la arquitectura y, por último, escribir un archivo de configuración con las variables definidas.

6.1.4. Codificación

Para aumentar el nivel de portabilidad, el script se debe codificar utilizando comandoss de Bourne Shell (sh) plano. Este hecho implica numerosas limitaciones, las más notorias, la inexistencia de tipos fuera de las cadenas de texto y la inexistencia de vectores.

En los siguientes apartados se hace da una perspectiva global de como funciona el script y se muestran los recursos de programación utilizados.

Lectura de opciones de usuario

La primera fase del script utiliza un for para recorrer el vector de argumentos pasados al *configure* y se basa en un switch para distinguir las opciones entre si y tratarlas como corresponda. Para conseguir que los casos del switch distingan las opciones independiente del valor que reciben (-cc=gcc y -cc=mpicc deben ser tratados por el mismo case) se utiliza, para cada opción con valor, la expresión regular -option=*.

Para todas las opciones que tienen como valor un ejecutable o una librería (mpiinc, mpilib, dyninstinc, dyninstlib, dwarf, with-binutils y with-doxygen) se consigue el valor de la opción con el comando sed de esta forma:

```
[value] =`echo $arg | sed -e 's/-[option]=//'`
```

El objetivo de la expresión pasada a sed es substituir (s) el patrón -[option] por nada, quedando aislado el valor, que es guardado en una variable. Una vez obtenido el valor, se comprueba que la ruta sea válida con la opción -d que determina si una cadena es un directorio.

Para el caso de los compiladores de c y c++ se compara el valor con una lista de compiladores válidos, de no encontrarse en la lista, se comprueba si es una ruta, si lo es, se deja solo el nombre utilizando sed y se vuelve a comparar con la lista.

Para mejorar la usabilidad se incluye la opción -help que muestra la lista de opciones posibles para *configure*.

Detección de la arquitectura del procesador

Por razones de productividad, la detección de la arquitectura del procesador se hace mediante el script *archfind* que provee el equipo de desarrollo de TAU, haciendo referencia al equipo de desarrollo original tal y como marca la licencia GPL bajo la que está distribuido.

Ya que entre las opciones está la selección de la arquitectura manualmente, si el usuario la ha utilizado, comprobamos si coinciden ambas arquitecturas, de no hacerlo se muestra un mensaje de advertencia y se utiliza la marcada por el usuario.

Definición de variables restantes

Para todas aquellas variables que no hayan sido definidas por el usuario se definen con valores por defecto, de no ser posible, se aborta el script y se informa del problema.

La arquitectura detectada se utiliza tanto para determinar el binario que corresponde a los compiladores como para determinar la ruta a la librería MPI y sus opciones.

Para el resto de librerías, en cambio, se utiliza una lista de lugares por defecto donde se deberían ubicar (/usr/lib, /usr/local/lib, etc), de no encontrarse en ninguna de esas rutas, se cierra el *configure* y se informa al usuario sobre la dependencia no satisfecha.

Confección archivo de configuración

Una vez definidas las variables se escriben sus valores en un archivo de texto llamado *make.configure* de la siguiente forma:

```
echo "DYNINST_LDIR  = $dyninstlib" >> make.configure
```

6.1.5. Prueba del instalador

Para el instalador se plantearon y codificaron una serie de casos de prueba en forma de batería, con el objetivo de probarlo en entornos distintos con facilidad. A continuación se especifican estos entornos y que casos de prueba se corrieron.

Entornos de prueba

Los entornos donde se corrió la batería de pruebas son los siguientes: Ubuntu 11.04 (32 bits), Ubuntu 11.04 (64 bits), Ubuntu 10.04 LTS (32 bits) y openSuse[12] 11.4 (32 bits). Cada uno de ellos preparado con la siguiente disposición de librerías, compiladores, etc.:

- Compiladores de c: cc, gcc.
- Compiladores de c++: g++,mpicxx (este se instala cuando se instala mpi)
- MPI
    - include: /usr/include (por defecto cuando lo instalas) y /opt/include (forzado)
    - lib: /usr/lib (por defecto cuando lo instalas) y /opt/lib (forzado)
- Dyninst: /usr/dyninstAPI, /opt/dyninstAPI
- libdwarf: /opt/dwarf (aparte del que esta en dyninst)
- libelf: /usr/include/libelf.h
- libiberty: /usr/include/libiberty.h

Resumen de casos de prueba

Las siguientes fichas describen los casos de prueba llevados a cabo.

Caso de prueba: 1	Nombre: Todo automático
Llamada: ./configure	
Salida esperada: cc = gcc c++ = g++ mpiinc =/usr/include mpilib = /usr/lib dyninstinc = /usr/dyninstAPI/src/include dyninstlib = /usr/dyninstAPI/src/[depende del so]/lib dwarf = /usr/dyninstAPI/src/[depende del so]/lib elf: /usr/include/libelf.h libiberty: /usr/include/libiberty.h	

Caso de prueba: 2	Nombre: Todo forzado
Llamada: ./configure -cc='cc' -c++='mpicxx' -mpiinc='/opt/include' -mpilib='/opt/lib' -mpilibrary=-lmpi_r -dyninstinc='/opt/dyninstAPI/src/include' -dyninstlib='/opt/dyninstAPI/src/[dynarch]/lib' -dwarf='/opt/dwarf/lib'	
Salida esperada: cc = cc c++ = mpicxx mpiinc =/opt/include mpilib = /opt/lib -mpilibrary=-lmpi_r dyninstinc = /opt/dyninstAPI/src/include dyninstlib =/opt/dyninstAPI/src/[dynarch]/lib dwarf = /opt/dwarf/lib elf: /usr/include/libelf.h libiberty: /usr/include/libiberty.h	

Caso de prueba: 3	Nombre: Entrada desconocida
Llamada: ./configure -invalida=invalida	
Salida esperada: Igual que automático (desecha la opción inválida)	

Caso de prueba: 4	Nombre: Compiladores, librerías inexistentes, etc
Llamada: ./configure -mpiinc='invalid/path' (por ejemplo)	
Salida esperada: Mensaje de error y salida	

Caso de prueba: 5	Nombre: Opción duplicada
Llamada: ./configure -dyninstinc='/usr/dyninstAPI/src/include' -dyninstinc='/opt/dyninstAPI/src/include'	
Salida esperada: Se queda la última	

Para más información sobre como se codificaron los casos de prueba, consultar la memoria de Toni Pimenta. La Tabla 6.1 muestra el resultado de la ejecución de los casos de prueba:

	Ubuntu 11.04 (32 bits)	Ubuntu 11.04 (64 bits)	Ubuntu 10.04 (32 bits)	OpenSuse 11.4 (32 bits)
Prueba 1	Correcto	Correcto	Correcto	Correcto
Prueba 2	Correcto	Correcto	Correcto	Correcto
Prueba 3	Correcto	Correcto	Correcto	Correcto
Prueba 4	Correcto	Correcto	Correcto	Correcto
Prueba 5	Correcto	Correcto	Correcto	Correcto

Tabla 6.1. Resultados de casos de prueba

6.1.6. Prueba de aceptación

Con el objetivo de averiguar el grado de usabilidad adquirido con esta característica se hizo un pequeño test de aceptación. El grado de satisfacción fue aceptable, pero se descubrió que era común el uso de rutas relativas en las opciones, característica que no era esperada.

Para soportarla se creo un script llamado *toAbs* para preprocesar las rutas, cada vez que se lee el valor de una opción se pasa llama a este script para procesarla.

El funcionamiento del script es sencillo, si se trata de una ruta absoluta, la retorna, si no, concatena el directorio de trabajo y retorna el resultado. La distinción entre absoluto y relativo la hace el primer carácter (si es una '/' es absoluta), pero a no tener acceso a vectores en Bourne shell esta distinción se debe hacer por medio de una expresión regular:

```
path_abs =`echo $2 | sed -ne '/^\// p'`
```

6.2. Módulo de cerrado

6.2.1 Especificación

El objetivo de este módulo es cerrar la aplicación de forma controlada, lo cual implica que se ha de cerrar el módulo Analyzer y cada uno de los módulos AC en los otros nodos, pero teniendo en cuenta que este proceso ha de ser llevado a cabo de forma centralizada.

En el proceso de cerrado, el módulo Analyzer debe liberar la memoria tanto de la DTAPI como la del tunlet activo; por su parte, el modulo AC debe descargar DMLib de la aplicación objetivo.

6.2.2. División del trabajo

En el desarrollo de esta característica trabajan dos miembros del equipo de desarrollo, Rodrigo Echeverría, autor del presente documento, y Toni Pimenta. Cada uno se encargara del diseño, la implementación y el test de la característica en el módulo asignado en la etapa de implantación de la metodología, a saber, Rodrigo trabaja sobre Analyzer y Toni sobre AC.

6.2.3. Diseño

Puesto que el módulo Analyzer mantiene una referencia al modelo de aplicación y, por lo tanto, tiene encapsulada la información sobre en que nodos se está ejecutando, el módulo de cerrado se puede centrar fácilmente en Analyzer. De esta forma el problema inicial se traduce en habilitar una entrada de datos en Analyzer para que el

usuario pueda indicarle cuando parar y, al recibir esta notificación, enviar una señal de parada a los AC registrados y, posteriormente, detenerse a si mismo.

El diseño del módulo de cerrado tanto para el caso del módulo Analyzer como el del módulo AC consiste, básicamente, en resolver tres problemas:

- Establecer un medio de comunicación y un protocolo común.
- Esperar la señal de parada (por parte del usuario en caso de Analyzer y por parte de Analyzer en el caso de AC) sin bloquear el flujo normal de ejecución.
- Parar el flujo principal de ejecución de forma estable.

El primer problema se puede resolver mediante el uso de sockets, que son un recurso que pone al alcance el sistema operativo para comunicar procesos incluso cuando estos se están ejecutando en nodos diferentes. Además, puesto que MATE ya requería de un sistema de comunicaciones ya existen clases que implementan y abstraen toda la estructura de datos necesaria para la comunicación a través de sockets.

La solución general al problema consiste en un esquema cliente-servidor implementado mediante sockets en el que cada AC actúa como servidor a la espera de peticiones del Analyzer, que hace el papel de cliente. De hecho, desde un punto de vista estricto, cada AC solo recibe una petición, la de cerrado.

Respecto al segundo problema, se puede solucionar también mediante otro recurso del sistema operativo los procesos ligeros o hilos de ejecución (threads, en inglés). Para poder esperar la señal de cerrado sin bloquear el flujo principal, basta con crear un hilo de ejecución secundario y delegar en él la responsabilidad. Este hilo se bloquea a la espera de la señal de cerrado y, al recibirla, emprende las acciones necesarias que, en el caso de Analyzer consisten en propagar la señal entre los ACs y, después, enviar una señal de cerrado al flujo principal para que emprenda acciones de limpieza y se cierre.

La Figura 6.1 muestra un esquema general de los flujos de ejecución tanto de Analyzer como de AC y detalla en que punto y por parte de quién son interrumpidos.

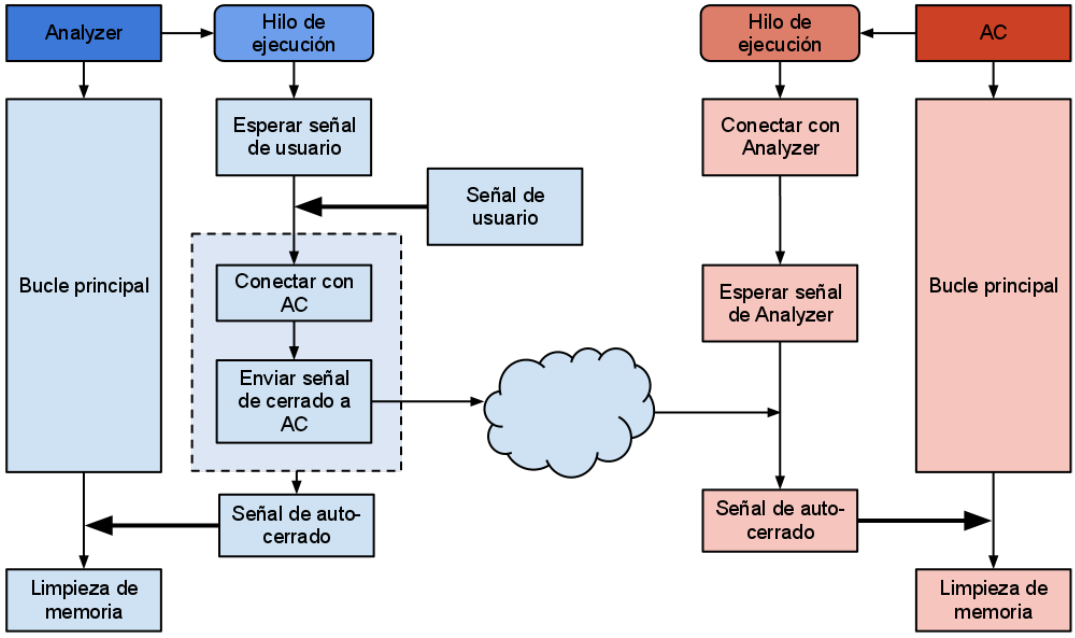

Figura 6.1. Ejecución del sistema de cerrado

La resolución del tercer problema (parar el módulo de forma estable) para el caso de Analyzer, requiere conocer como es su flujo de ejecución. La Figura 6.2 muestra un diagrama de secuencia UML que analiza dicho flujo.

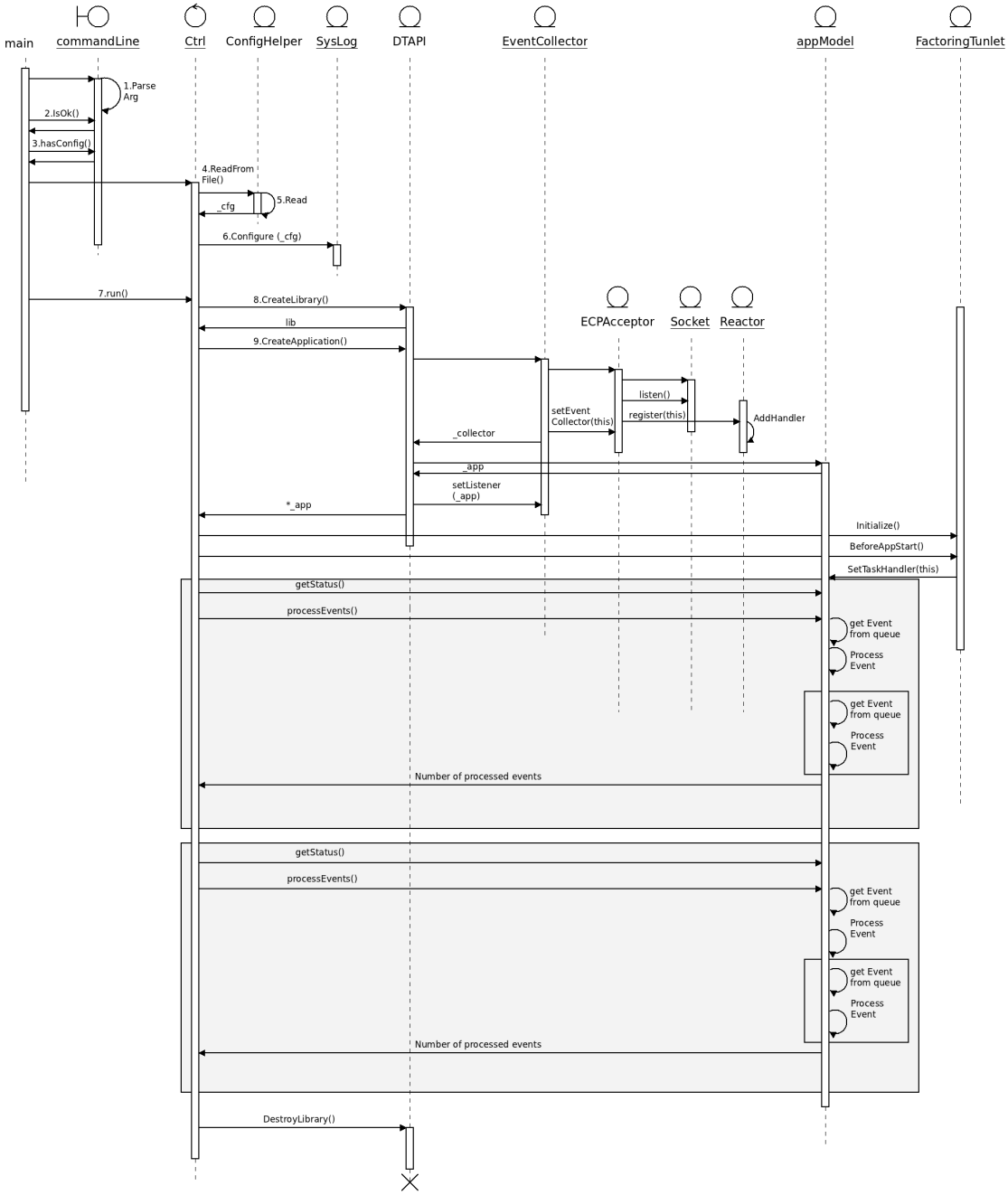

Figura 6.2. Diagrama de secuencia del módulo Analyzer

De este análisis se pueden extraer varias conclusiones respecto a la interrupción del flujo de ejecución:

Existe un breve periodo de de tiempo antes de que el modelo de aplicación sea creado, en este periodo intentar acceder a la lista de servidores donde están alojados los AC, provocaría una excepción de puntero nulo.

Después de ese periodo de tempo existe un lapso indefinido de tiempo hasta que los AC envían un mensaje para registrarse en la lista de Analyzer. Analyzer bloquea su flujo a la espera del primero de estos mensajes de registro (primer rectángulo gris). Durante este periodo el cerrado es muy delicado puesto que no se sabe si aún quedan ACs por registrarse y además Analyzer esta bloqueado por un semáforo.

El tercer periodo es el que corresponde al bucle principal de Analyzer (segundo rectángulo gris), en el Analyzer se queda en un bucle infinito a la espera de eventos. Estos eventos pueden corresponder a registro de nuevos nodos, a nodos que se eliminan o a eventos relacionados con la monitorización.

Para resolver el conflicto que se produce en las primeras fases debido a que, o bien no se ha creado el modelo de aplicación, o bien no se a registrado ningún AC aún, se puede controlar que el objeto que representa al modelo de aplicación o su miembro hosts (lista de nodos con un AC registrado) sean nulos, y, de serlo, basta con frenar Analyzer y notificar al usuario que no se paro ningún AC en el proceso.

El problema que se produce en la segunda fase debido a que Analyzer se bloquea a la espera del primer evento es más delicado, puesto que al estar bloqueado el flujo principal no recibe la señal de auto-cierre y no acaba su ejecución. Una solución posible es eliminar el bloqueo y hacer que espere al primer evento como a todos los demás (de forma no bloqueante). En principio esto no debería afectar a la correcta ejecución del programa, ya que que el primer evento es el registro del primer AC (que se trata de forma bloqueante) pero entre los siguientes eventos se registran los demás (de forma no bloqueante) y esto indica que el registro se puede hacer de esta última forma.

6.2.4. Codificación

La implementación del módulo de cerrado para el caso de Analyzer consiste en una clase denominada *ShutDownManager* en alusión a que es el punto central del sistema.

La clase ShutDownManager hereda de la clase activeObject (que implementa hilos de ejecución), de esta forma un objeto de la clase ShutDownManager constituye un hilo.

Los objetos de la clase ShutDownManager poseen dos miembros principales _isFinished y _app . _isFinished determina si la aplicación está acabada, el proceso principal tiene acceso a él, y lo comprueba al entrar en los bucles, si es verdadero rompe la ejecución y salta a la fase de limpieza y cerrado (ver figura x.y.z). _app guarda una referencia al modelo de aplicación que tiene el proceso principal. Cuando el proceso principal crea este modelo llama al método setApp con un puntero a éste; a partir de ese punto ShutDownManager tiene acceso a la lista de servidores en que la aplicación se está ejecutando.

El método principal de la clase se llama Run, que no es más que una reimplementación del heredado de la clase activeObject y que es la función que ejecuta el hilo. Este método consiste en hacer scanf de la entrada estándar (queda bloqueado hasta que el usuario escribe algo) y cuando el usuario escribe la letra 's', procede a recorrer la lista de servidores de _app, conecta un socket a cada uno de ellos y envía una señal de cerrado, una vez hecho esto define a cierto la variable _isFinished para que el proceso principal del propio Analyzer pase a la fase de limpieza y se cierre.

6.2.5. Prueba unitaria

El caso de prueba que se propone para el *ShutDownManager* es relativamente sencillo de plantear, pero complejo de codificar para que sea realmente una prueba unitaria *[JEN 08]*. El caso de prueba consiste en crear dos aplicaciones, la primera para simular el comportamiento de Analyzer (a partir de aquí, *driver*) en la que se crea un objeto de la clase ShurDownManager y, la segunda, que simule el comportamiento de un AC respecto a las comunicaciones.

La primera aplicación, el driver, es relativamente complejo de codificar, puesto que el grado de acoplamiento entre el objeto de la clase ShutDownMaager (a partir de este punto, sdm) y el Analyzer original es muy alto. El grado de acoplamiento alto se debe a que sdm coge la lista de servidores de la aplicación objetivo directamente del modelo de aplicación que se encuentra en Analyzer (mantiene una referencia llamada _app, como se comentaba en la sección anterior).

Este hecho se traduce en que simular Analyzer implica que, para que la prueba sea realmente unitaria y se puedan forzar valores, se tengan que crear clases que produzcan objetos de tipo aplicación y de tipo host (servidor). Estas clases falsas han de presentar una interfaz similar a las originales (de hecho, se han de mantener las signaturas exactas de las clases originales, al menos para las funciones que se usen), pero deben permitir forzar su funcionalidad. Por ejemplo, en el caso de la aplicación, esta debe mantener un objeto _hosts correspondiente a la lista de servidores de la aplicación, pero en lugar de llenarlo cuando recibe un mensaje de registro (como hace Analyzer) ha de permitir acceso y modificación manual a dicha lista.

Una vez codificadas las clases que simulan a Application y a Host, la programación del driver consiste en un proceso principal que crea un sdm y un modelo de aplicación, se asocia este modelo al sdm y,luego, se llena su la lista de servidores manualmente utilizando los valores convenientes de prueba.

La segunda aplicación consiste en un proceso principal que crea una conexión y se pone a la espera de peticiones (tal y como harían los AC).

La ejecución del caso de prueba consiste en ejecutar ambas aplicaciones, pasándole al driver el host de la segunda, hacer la entrada del código de salida ('s') en la primera y observar si la señal llega correctamente al otro punto. La Tabla 6.2 resume los resultados de este caso de prueba.

Entrada	Resultado
“127.0.0.1” (localhost)	Correcto
localhost	Incorrecto
“192.168.x.y” (otro miembro de la red)	Correcto
NombreNodo	Incorrecto

Tabla 6.2. Resultados de la ejecución de casos de prueba

En la Tabla 6.2 se puede observar que el mecanismo funciona, pero solo si la entrada es la ip de la máquina, de hecho, el fallo que produce el uso del nombre del nodo es sistemático y comporta el aborto de la aplicación entera.

Esta particularidad hizo sospechar que el problema estaba en el procesado de la dirección para crear el socket (classe Adress de Common) , que es polimórfica: si es una ip se crea directamente, si es un nombre se traduce con la llamada a sistema getHostByName(). En este punto el motivo de fallo se reduce a dos candidatos un fallo de getHostByName() o bien un fallo en la creación de la estructura de datos que representa al servidor en el Socket. Después de probar que getHostByName() se ejecutaba correctamente, se concluye que la causa del problema es la segunda, y, a base de aislar el problema se detecta la llamada que lo produce:

```
::strncpy  ((char*)&_addr.sin_addr,pHostEntry->h_addr_list
[0], pHostEntry->h_length);
```

Esta función copia la ip del servidor en la estructura de datos que utiliza el socket, pero, utiliza para ello la función strncpy, que no es apropiada. En casos habituales, se utiliza la función bcopy (byte copy), que copia a más bajo nivel y presenta mejores resultados.

```
::bcopy    ((char*)&_addr.sin_addr,pHostEntry->h_addr_list
[0], pHostEntry->h_length);
```

Una ver realizado este cambio, los cuatro casos de prueba funcionan perfectamente.

7. Conclusión

En este capítulo se resume el trabajo mostrado haciendo referencia a la consecución de objetivos y a las posibles lineas de trabajo futuras.

Este proyecto representa un paso adelante en calidad de software en el proyecto MATE. En el se han propuesto tres lineas de trabajo para mejorar y apoyar diferentes aspectos del mismo, en este sentido se puede concluir que es un proyecto exitoso tanto a nivel de lo expuesto en esta memoria como en global.

La primera linea de proyecto, el desarrollo y la implantación de una metodología de trabajo se ha concluido correctamente, por una parte, se han creado una serie de especificaciones que servirán como punto de partida para los nuevos desarrolladores del proyecto y, por la otra, se ha aplicado dicha metodología al proyecto existente.

La implantación de la metodología implica, entre otras cosas, un salto cualitativo muy grande en aspectos de mantenibilidad, puesto que se ha creado, por primera vez, todo el conjunto de documentación para desarrolladores, imprescindible para la incorporación efectiva de nuevos miembros al equipo.

La segunda linea de proyecto, la implantación de un entorno de desarrollo, también concluye satisfactoriamente puesto que se ha llegado a un entorno capaz de manejar el proyecto de forma íntegra y de aportar herramientas de apoyo de todo tipo (control de versiones, construcción, documentación, etc.).

La tercera linea de proyecto, la creación de nuevas características, ha logrado implementar una primera fase de dos de los módulos secundarios más imprescindibles para una aplicación de este calibre, el modulo instalador y el modulo de cerrado. El primero de ellos, en particular, supone un paso enorme en conceptos de portabilidad y distribución.

Por otra parte y, debido a que este es un proyecto trascendental, cabe destacar que durante el proyecto han quedado patentes nuevas necesidades, aspectos que se podrían ampliar y características que se podrían mejorar. A continuación se plantean algunas

lineas a futuro relacionadas con el presente proyecto:

- Especificación de codificación de tunlets: esta guía se vuelve un indispensable de cara a la experimentación con nuevos modelos de mejora del rendimiento de aplicaciones paralelas. Queda fuera del alcance de un proyecto como este debido a la necesidad de conocimientos específicos sobre computación de altas prestaciones.
- Guía de estilo de prueba unitaria y prueba en general: en sistemas complejos (como MATE) la prueba de características es una necesidad básica, pero normalmente compleja de cara a hacer pruebas verdaderamente significativas.
- Manual de migración de BDD de Redmine: esta característica sería un buen complemento a la automatización del proceso de instalación, puesto que, en la mayoría de situaciones que impliquen reinstalación, se requerirá una migración de datos para no perder el proyecto.
- Base de datos de documentación e integración con Redmine: uno de los primeros problemas que surgieron a partir de la codificación de documentos es la necesidad de conocer que código les corresponde (para mantener la unicidad). Nosotros paliamos el problema creando una micro base de datos en un directorio concurrente, pero esta solución tiene como problema que no esta asociada al entorno de desarrollo y, puesto que este pretende ser una solución todo-en-uno, la base de datos se tendría que integrar en Redmine.
- Instalador basado en autotools: el uso de autotools implicaría un paso adelante en cuestiones de portabilidad y estandarización del módulo. Si bien la solución planteada es totalmente funcional, una implementación basada en autotools sería más fácil de mantener.
- Prueba de integración de sistema de cerrado: queda pendiente la prueba de integración del sistema de cerrado por cuestiones técnicas. Esta prueba es imprescindible para discernir si el sistema es totalmente funcional, aunque las pruebas unitarias ya apunten a ello.

Bibliografía

[CAY 07] *Paola Caymes*, "Extending the Usability of a Dynamic Tuning Environment", Phd. Thesis. Universitat Autònoma de Barcelona. 2007.

[FIL 07] *Pablo R. Fillottrani*, (2007). "Calidad en el desarrollo de software. Modelos de calidad". Depto. Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur [Online]. http://www.cs.uns.edu.ar/~prf/teaching/SQ07/clase6.pdf [5 Abril 2011]

[GIE 11] *Dr. Holger Giese*,(Fecha no especificada)."Software Quality Assurance: Introduction". University of Paderborn Software Engineering Group [Online].

http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Vorlesungen/SoftwareQualityAssurance/WS0405/SQA-I.pdf [20 Marzo 2011]

[IBM81] IBM, (1981). "Implementating Software Inspections", Notas IBM Systems Sciences Institute, IBM Corporation.

[JEN 08] *Nick Jenkins*, (2008). "A Software Testing Primer. An Introduction to Software Testing". Nick Jenkins prose [Online] www.nickjenkins.net/prose/testingPrimer.pdf [Mayo 2011]

[LUQ 00] *Antonio Luque Estepa*,(2000). "Herramientas de desarrollo bajo Linux". Escuela superior de Ingenieros de Sevilla [Online]. http://woody.us.es/~aluque/doc/herram.pdf [Abril 2011]

[MAR 11] *Noel De Martin*. "Aplicación de la ingeniería del software sobre la herramienta MATE: Common y DMLib". Universitat Autonoma de Barcelona (2011)

[MOR 04] *Anna Morajko*, "Dynamic Tuning of Parallel/Distributed Applications",

Phd. Thesis. Universitat Autònoma de Barcelona. 2004.

[PIM 11] *Toni Pimenta*. " Aplicación de la ingeniería del software sobre la herramienta MATE:Application Controller". Universitat Autonoma de Barcelona (2011)

[VEI 07] *Alejandro Veiga*, (2007)."Arquitecturas Paralelas II (MIMD)".Facultad de ingeniera.Universidad Nacional de La Plata [Online].http://electro.fisica.unlp.edu.ar/arq/transparencias/ARQII_08-MIMD.pdf [16 Septiembre 2011]

[WIKI 11] *Colaboradores de Wikipedia*,(Setiembre 2011). "High-performance computing". Wikipedia, The Free Encyclopedia [Online].http://en.wikipedia.org/w/index.php?title=High-performance_computing&oldid=448675937 [29 Agosto 2011]

Enlaces web

[1] MPI. www.mpi-forum.org [Accedido Noviembre 2010]

[2] Dyninst. www.dyninst.org [Accedido Enero 2011]

[3] Apache HTTP Server. http://www.apache.org/ [Accedido Enero 2011]

[4] Redmine. http://www.redmine.org/ [Accedido Enero 2011]

[5] Trac. http://trac.edgewall.org/ [Accedido Enero 2011]

[6] Subversion. http://www.apache.org/ [Accedido Enero 2011]

[7] Buildbot. http://trac.buildbot.net/ [Accedido Enero 2011]

[8] Doxygen. http://www.stack.nl/~dimitri/doxygen/ [Accedido Enero 2011]

[9] Eclipse. http://www.eclipse.org/ [Accedido Febrero 2011]

[10] Phusion passenger. http://www.modrails.com/ [Junio 2011]

[11] TAU. http://www.cs.uoregon.edu/Research/tau/home.php [Abril 2011]

[12] OpenSuse. http://es.opensuse.org/ [Accedido Abril 2011]

[13] Ubuntu. http://www.ubuntu.com/ [Accedido Abril 2011]

Índice de anexos